NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SCHEDULING AND PROTOTYPING OF DISTRIBUTED REAL-TIME SYSTEMS (AN APPROACH USING JINI/JAVASPACES)

by

Tolga DEMIRTAS

March 2002

Thesis Advisor:	Man-Tak Shing
Second Reader:	Joseph Puett

Approved for public release; distribution is unlimited

Report Documentation Page

Report Date	Report Type	Dates Covered (from... to)
29 Mar 2002	N/A	-

Title and Subtitle Scheduling and Prototyping of Distributed Real-Time Systems (An Approach Using JINI/JAVASAPCES)	**Contract Number**
	Grant Number
	Program Element Number
Author(s) Demirtas, Tolga	**Project Number**
	Task Number
	Work Unit Number
Performing Organization Name(s) and Address(es) Naval Postgradaute School Monterey, California	**Performing Organization Report Number**
Sponsoring/Monitoring Agency Name(s) and Address(es)	**Sponsor/Monitor's Acronym(s)**
	Sponsor/Monitor's Report Number(s)
Distribution/Availability Statement Approved for public release, distribution unlimited	
Supplementary Notes The original document contains color images.	
Abstract	
Subject Terms	
Report Classification unclassified	**Classification of this page** unclassified
Classification of Abstract unclassified	**Limitation of Abstract** UU
Number of Pages 241	

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 2002	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE: Scheduling and Prototyping of Distributed Real Time Systems (An approach using Jini/JavaSpaces)	5. FUNDING NUMBERS
6. AUTHOR(S) **Tolga Demirtas**	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT ***(maximum 200 words)***

The major difference between single processor and distributed processors scheduling is that, in addition to deciding which order to execute tasks, distributed processors' scheduling algorithms must also decide which processor the task should run on. Moreover, these algorithms must also take into consideration practical network issues like transmission delay, loss of messages, and synchronization in the absence of a global clock. This thesis proposes a formal model to capture these network constraints and develops a proxy-based network buffer technique to support the inter-process communication for the user-defined distributed real-time systems prototypes generated by the Distributed Computer Aided Prototyping System (DCAPS). The proxy-based technique builds on the Jini/JavaSpaces infrastructure. We have conducted several experiments to measure the response time of inter-process communication via JavaSpaces. We have demonstrated the effectiveness of the proxy-based technique by creating an executable prototype of a user-defined distributed real-time system specification.

14. SUBJECT TERMS Computer Aided Prototyping, Distributed Real-Time Systems, Jini, JavaSpaces, Scheduling, Prototyping.	15. NUMBER OF PAGES 241
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SCHEDULING AND PROTOTYPING OF DISTRIBUTED REAL TIME SYSTEMS (AN APPROACH USING JINI/JAVASPACES)

Tolga Demirtas
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1996

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2002

Author: Tolga Demirtas

Approved by: Man-Tak Shing, Thesis Advisor

Joseph Puett, Second Reader

C. S. Eagle, Chairman
Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Scheduling is one of the basic issues in building real-time applications on a distributed computing system. A distributed computing system is typically modeled as a collection of processes interconnected by a communication network. For real-time applications, scheduling is needed to meet applications timing constraints.

The major difference between single processor and distributed processors scheduling is that, in addition to deciding which order to execute tasks, distributed processors' scheduling algorithms must also decide which processors the task should run on. Moreover, these algorithms must also take into consideration practical network issues like transmission delay, loss of messages, and synchronization in the absence of a global clock. This thesis proposes a formal model to capture these network constraints and develops a proxy-based network buffer technique to support the inter-process communication for the user-defined distributed real-time systems prototypes generated by the Distributed Computer Aided Prototyping System (DCAPS). The proxy-based technique builds on the Jini/JavaSpaces infrastructure. We have conducted several experiments to measure the response time of inter-process communication via JavaSpaces. We have demonstrated the effectiveness of the proxy-based technique by creating an executable prototype of a user-defined distributed real-time system specification.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank to my second reader, LTC Joseph Puett, for all his help and advice.

I would like to express my gratitude to my thesis advisor, Professor Shing, for all his support, guidance, and confidence. He always made himself available whenever I have needed his help. I appreciate his extraordinary knowledge of this field. It was fortunate for me to work with him.

I would also like to thank my mother, Sevim, for her devotion, love, and support throughout my life. I owe her everything. She was always with me even though I have been thousands of miles away from home.

Finally, I would like to thank God for helping me to advance one more step in my life and my career.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

This thesis deals with the area of computer-aided distributed real-time systems development and investigates the technology for generating distributed real-time systems from formal specifications for a given target platform. In this thesis, we primarily focus on the inter-process communication in the distributed real-time systems for heterogeneous networks. This thesis also provides an implementation of a distributed real-time system to generate code for the prototype to run on the target platform defined by the hardware model.

We investigate a new technology, Jini/JavaSpaces, to use for inter-process communication in development of distributed real-time systems using the Distributed Computer Aided Prototyping System (DCAPS). This chapter presents the basic concepts of distributed real-time systems and DCAPS. Issues related to distributed systems are discussed in Chapter II. Chapter III explains the Jini/JavaSpaces technology in depth and explores the characteristics and advantages of using this technology in the development of distributed real-time systems. Chapter IV discusses the main principles of the Prototyping System Description Language (PSDL) used in DCAPS, and explains how we can use the Jini/JavaSpaces technology to support these principles for inter-process communication. Chapter V presents an example implementation based on the solutions we developed. The experiments conducted to measure the response time of Jini/JavaSpaces services for inter-process communication are explained in detail in Chapter VI. We conclude the work done in Chapter VII.

This thesis does not provide detailed information about DCAPS and PSDL. We assume that the reader has a strong background in DCAPS and PSDL.

A. DISTRIBUTED SYSTEMS

In the past few years, the computing landscape has changed dramatically. Many devises, such as hand-phones, Personal Device Assistants (PDAs), and Internet Terminals, etc., have been enhanced with network capabilities to leverage the benefits

that new communication technologies have brought. Industrial companies and military services often operate or communicate via the Internet in a broad area to streamline operations and cut expenditures. Devices and software components have become more tightly coupled, cooperating together in a distributed system to accomplish common goals.

We can define a distributed computing system as a system of multiple autonomous processing elements, cooperating for a common purpose or to achieve a common goal. Figure 1.1 shows a distributed system. “Distributed Computing” is all about designing and building applications as a set of processes that are distributed across a network of machines and work together as an ensemble to solve a common problem [FHA99].

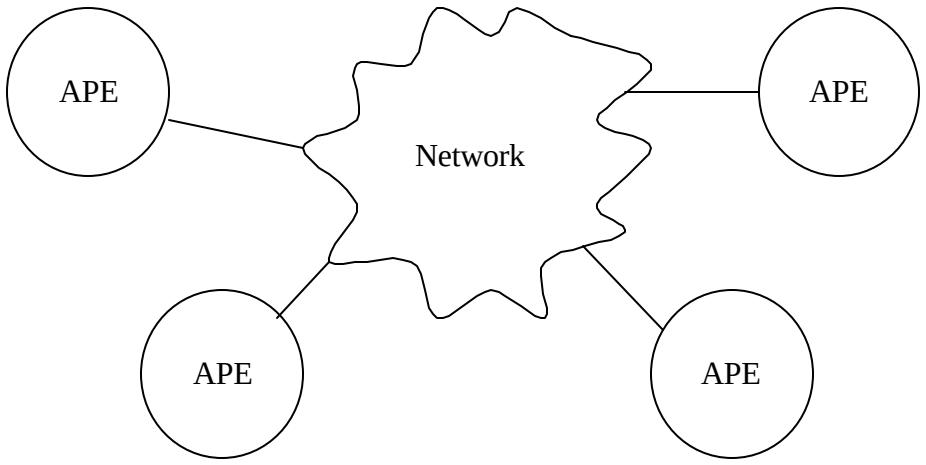

Figure 1.1 A Distributed System with Autonomous Processing Elements (APE)

If a distributed system consists of a collection of autonomous processing elements, then it is considered to be a “loosely-coupled” system. If a distributed system consists of multiple processing units, sharing a single memory and address space, then it is considered to be a “tightly-coupled” system [CAA98]. We focus on the architecture

and design of "loosely-coupled" distributed systems in this thesis. Because of this, we will use the term "distributed systems" to mean loosely-coupled distributed systems.

B. REAL-TIME SYSTEMS

Systems that must correspond with each other as fast as possible are sometimes informally called "real-time systems". This objective is rarely defined precisely, and is often interpreted as minimizing response time with respect to some performance measure, such as the average delay [LUQ93].

Another definition of real-time systems is provided by [LAP93]: "A real-time system is a system that must satisfy explicit (bounded) response time constrains or risk severe consequences, including failure."

A real-time system is one that has performance deadlines on its computations and actions. Real-time systems are often embedded, meaning that the computational system exists inside a larger system, with the purpose of helping that system to achieve its overall responsibilities. Performance requirements are most commonly specified in terms of deadlines. A deadline is either a point in time (time-driven) or a delta-time interval (event-driven) by which a system action must occur [DOU01].

We can separate real-time systems into three major groups according to their timing constraints as shown in Figure 1.2. A real-time system, which is well defined and has responses that occur within the specified deadlines for all tasks, is called a "hard" real-time system. Timeliness is essential to correctness in hard real-time systems. A missed deadline constitutes an erroneous computation and a system failure. In hard real-time systems, late data is at best worthless data and at worst, bad data. A real-time system, which may be constrained simply by average execution time or by more complex constraints, is called a "soft" real-time system. Missing some deadlines, by some amount, under some circumstances, is acceptable for soft real-time systems. In soft real-time systems, late data may still be good data. Some systems have both soft and hard deadline performance constraints. These types of real-time systems are called "firm" real-time systems. These so-called firm deadlines arise when individual deadlines may be missed,

as long as two things occur. First, a sufficient average performance maintained at all times. Second, each deadline must be met no later than a certain time. A schedulable system is one that can be guaranteed to meet all its performance requirements [DOU01].

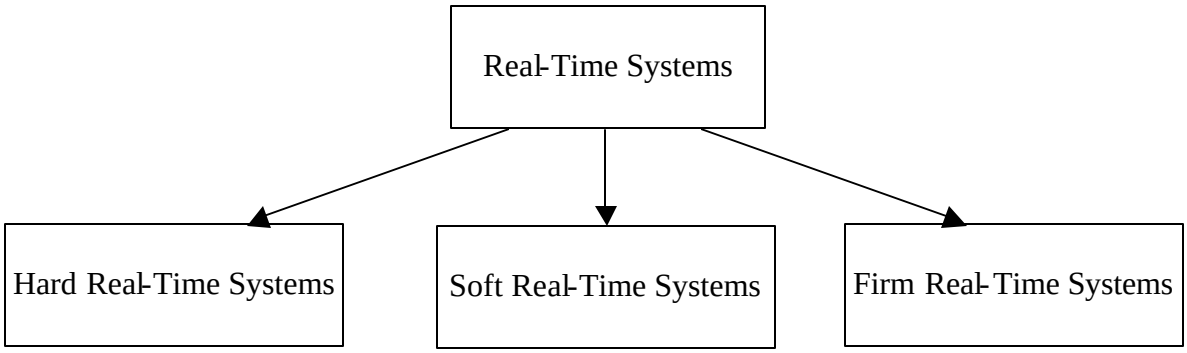

Figure 1.2 Real-Time Systems

One of the major differences between a hard real-time system and a conventional system is that the application software must meet its deadlines even under worst-case conditions. Large scale, parallel and distributed, hard real-time systems are important to both civilian and military applications. Examples of hard real-time systems include air traffic control systems, controls for automated factories, telecommunication systems, space shuttle avionics systems, and C3I systems. Hard real-time software systems are often embedded in larger systems, performing critical functions [LAS96].

C. RAPID PROTOTYPING AND CAPS

Over the past years, the demand for hard real-time and embedded systems has increased. These kinds of systems generally have strict requirements on their accuracy, safety and reliability. Feasible requirements for these kinds of systems are difficult to formulate, understand and meet without extensive prototyping. Computer aid is the key to rapid construction, evaluation and evolution of such prototypes [LUQ93].

The Distributed Computer Aided Prototyping System (DCAPS) is an integrated software development environment which has been developed at the Naval Postgraduate School for rapid prototyping of hard real-time embedded software systems, such as

missile guidance systems, space shuttle avionics systems, software controllers for a variety of consumer appliances and military Command, Control, Communication and Intelligence (C3I) systems [LUQ92]. DCAPS supports rapid prototyping and automatic generation of source code based on designer specifications in an evolutionary software development process [LBS00].

Rapid prototyping can be used to reduce the risk of producing systems that do not meet the customer needs [LUQ93]. Rapidly constructed prototypes are used to help both the developers and their customers visualize the proposed system and assess its properties in an iterative process.

The heart of DCAPS is the Prototyping System Description Language (PSDL). PSDL serves as an executable prototyping language at the architecture level and has special features for real-time system design. PSDL is a language for describing prototypes of real-time software systems. It is most useful for requirements analysis, feasibility studies, and the design of large embedded systems. PSDL has facilities for recording and enforcing timing constraints, and for modeling the control aspects of real-time systems using nonprocedural control constraints, operator abstractions, and data abstractions. PSDL has been designed for use with an associated prototyping methodology. PSDL prototypes are executable if supported by a software base containing reusable software components in an underlying programming language [LBY88].

Building on the success of the earlier versions of DCAPS, the DCAPS uses PSDL for specification of distributed systems and automates the generation of interface codes with the objective of making the network transparent from the developer's point of view [LBS01]. DCAPS also targets the heterogeneous distributed system development.

DCAPS supports an iterative prototyping process characterized by exploratory design and extensive prototype evolution [LAS96]. Figure 1.3 shows the DCAPS Rapid Prototyping Environment.

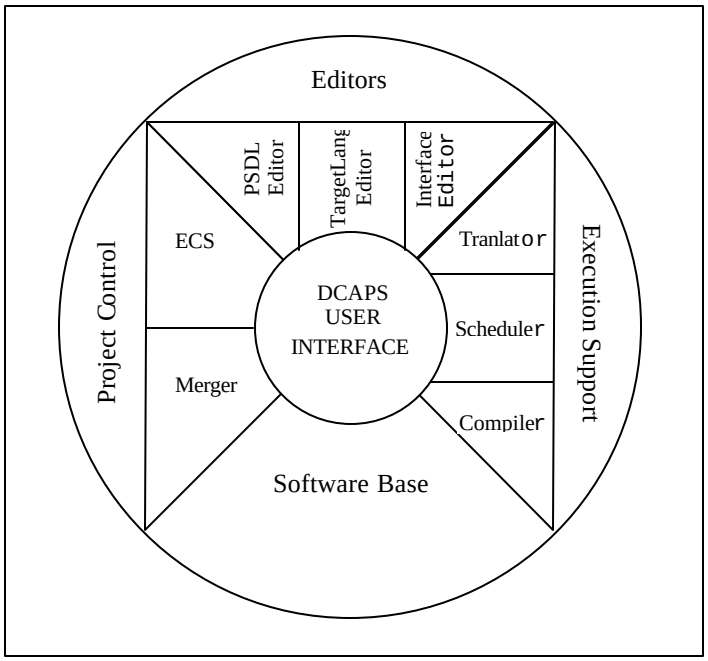

Figure 1.3 DCAPS Rapid Prototyping Environment

An automated prototyping methodology helps software engineers and end-users to validate functional requirements. It also helps them to verify design decisions and specifications early in the development phase using rapid prototyping.

There are four major stages in the DCAPS rapid prototyping process: software system design, construction, execution, and requirements evaluation/modification. Figure 1.4 shows the evolutionary prototyping process.

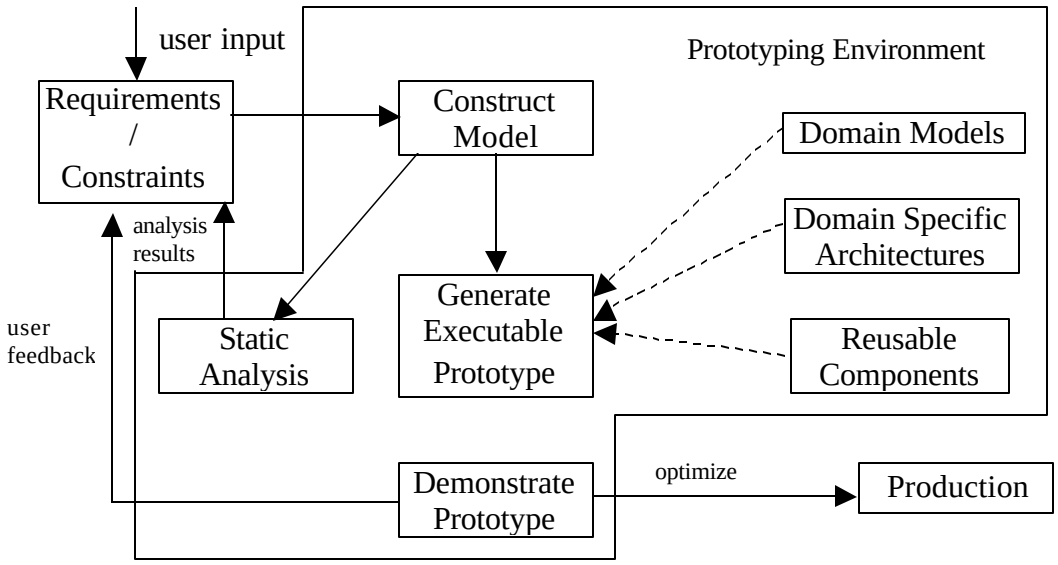

Figure 1.4 Evolutionary Prototyping Process

The initial prototype design starts with an analysis of the problem and a decision about which parts of the proposed system are to be prototyped. Requirements for the prototype are then generated, either informally (e.g., natural language) or in some formal notation.

After requirements analysis, the designer uses the DCAPS PSDL editor to draw dataflow diagrams annotated with nonprocedural control constraints as part of the specification of a hierarchically structured prototype, resulting in a preliminary top level design, free from programming level details. The designer may continue to decompose any software module until its components can be realized via reusable components drawn from the software base or realized with new atomic components. This prototype is then translated into the target programming language for execution and evaluation. Debugging and modification utilize a design database that assists the designers in managing the design history and coordinating change, as well as other tools shown in Figure 1.3 [LBM00].

D. SUMMARY

In this chapter, we presented “distributed systems” and “real-time systems” in general. We introduced “rapid prototyping” and pointed out that the computer aid is the key to rapid construction, evaluation, and evolution of such prototypes. We also introduced an integrated software development environment, DCAPS, which supports an iterative, rapid prototyping process.

II. ISSUES RELATED TO DISTRIBUTED SYSTEMS

This chapter discusses various issues related to the development of distributed systems. We first explain the benefits of distributed systems over standalone systems. Second, we discuss the challenges of distributed systems such as latency, synchronization, and fault tolerance. We then discuss the modeling strategies to build distributed systems and their advantages and disadvantages.

A. BENEFITS OF DISTRIBUTED SYSTEMS

Standalone systems strictly depend on the resources of the local environment in which they operate. Because of this limitation, distributed systems provide more benefits than standalone systems. By building distributed systems, we can make use of different resources, which are loosely coupled and have different capabilities. For example, if our resources are not sufficient to build an application, we can build a distributed system, which can access and make use of remote resources, to achieve our goal. Basic areas where distributed systems provide more benefits than standalone systems are [FHA99]: Performance, Scalability, Resource Sharing, Fault Tolerance and Availability, and Elegance.

1. Performance

A single CPU is limited by its speed. If we want to optimize our application and require better performance, the only thing we can do is to add another processor. Many problems can be decomposed into smaller ones. We can distribute these smaller problems over one or more processors to be computed in parallel. In principle, it appears that the more processors we have, the more jobs get done. In reality this is not the case. Continuously adding processors rarely results in perfect efficiency upgrade because of communication overhead and because tasks are seldom perfectly partitionable. Nevertheless, for many problems, this method (adding more processors) can reduce running time.

2. Scalability

When we write a standalone application, our computational power is limited to the power and resources of a single machine. If instead, we build a distributed application, we not only improve performance but we also create a scalable application. This means that if the existing machines cannot solve the problem, we can add additional machines without redesigning our application. We can scale our application according to the size of the problem.

3. Resource Sharing

Some computational resources may be expensive or unavailable for local access. We can support and coordinate remote access to such resources by constructing a distributed system. For effective sharing, each resource must be managed by a program that offers a communication interface enabling the resource to be accessed, manipulated, and updated reliably and consistently [CDK96].

4. Fault Tolerance and Availability

Standalone applications have little or no tolerance for failure. If a standalone application fails, it terminates and remains unavailable until it is restarted. Distributed systems can tolerate a limited amount of failure because they are made up of multiple, independent processes. Distributed systems can reduce “down-time” and maximize availability if they are designed carefully.

5. Elegance

For many problems, distributed systems provide the most natural and easy software solution. Many problems can be expressed as a dynamic relation of processes, which work asynchronously and communicate with each other. We can also define the world as a distributed system. For example, instructing a single worker to sequentially assemble a car is an inefficient approach. This worker’s task would be overly complex

and hard to maintain. Instead, we can divide the job into smaller parts and distribute them over multiple workers. This approach reduces the complexity of worker's tasks and makes it easier to maintain the system.

B. CHALLENGES OF DISTRIBUTED SYSTEMS

Despite their benefits, distributed systems are notoriously difficult to design, build and debug. The distributed environment introduces many complexities unencountered when writing standalone applications. The most obvious complexity is the variety of the machine architectures and software platforms over which a distributed application must execute.

The existence of a networked environment also presents many challenges beyond heterogeneity. Latency, Synchronization, and Partial Failure are the basic areas where distributed systems present challenges [FHA99].

1. Latency

Communication over a network can take a long time relative to the speed of processors. This time lag is called latency. Latency is typically several orders of magnitude greater than communication times between local processes on the same machine.

2. Synchronization

Communication is required but not sufficient to accomplish tasks in a distributed system. Distributed processes must also synchronize their actions. For example, a distributed algorithm might require processes to work in lock step in order to complete one phase of an algorithm before proceeding to the next phase. Processes also need to synchronize (essentially, wait their turn) in accessing and updating shared data. Synchronizing distributed processes is challenging, since the processes are truly asynchronous.

Another important point in synchronization is the need for the equivalence of a "global" clock, which may have significant impact on the performance of a distributed system.

3. Partial Failure

Partial failure is another important challenge of distributed systems. The longer an application runs and the more processes it includes, the more likely it is that one or more components will fail or become disconnected from the network. It is important to design distributed systems to be able to recover gracefully in the face of partial failures.

As mentioned, fault tolerance is an important aspect of distributed systems. It is important to carefully design distributed systems to recover from partial failures in order to gain the advantages of fault tolerance.

C. THE SEVEN FALLACIES OF DISTRIBUTED COMPUTING

Distributed systems introduce additional challenges than those of standalone systems. The network is the source of these new challenges. We already discussed some of these new challenges in this chapter (e.g., latency). Networks also fail in ways that standalone systems do not.

A lot of history of networked systems programming is about making the network transparent to the application programmers. Unfortunately, this simplification often turns out to be an oversimplification. This simplification tries to assume that a network connecting two software components will not affect the correctness of the program, only its performance.

The hardest part of building reliable distributed systems are not problems with packing data into portable forms nor invoking remote procedures; instead, the hardest part is the challenges introduced by the networked environment that cannot be ignored by the programmer. For example, the time required to access a remote resource may be orders of magnitude longer than accessing the same resource locally. Networked systems

are also susceptible to partial failures of computations that can leave systems in an inconsistent state [EDW01].

Computer scientist Peter Deutsch has written about what he calls “The Seven Fallacies of Distributed Computing” [EDW01]:

“Essentially everyone, when they first build a distributed application makes the following seven assumptions. All prove to be false in the long run, and all can cause big trouble and painful learning experiences.”

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology does not change
- There is one administrator
- Transport cost is zero

D. MODELLING DISTRIBUTED SYSTEMS

Perhaps the most important phase in the software development process is requirements analysis. It is difficult and expensive to correct design flaws and errors discovered late in a software process. Faulty, inconsistent and error-prone requirements result in flaws in the project.

Real-time systems have strict requirements for accuracy, safety and reliability. Also, it is difficult to formulate and understand these requirements.

As we discussed before, distributed systems are notoriously hard to build and design. If we are trying to build a distributed real-time system, then we must take into account all problems introduced by such systems.

It is a good idea to use prototyping to formulate and understand the requirements and design decisions of such systems. Analysis and measurement of prototype designs provide upper bounds on the time required to execute particular functions. Experiments with simulated environments provide information about the accuracies and response times required to keep external physical systems within desired operating constraints [LUQ93].

Rapid prototyping is useful to overcome these difficulties. The key idea of rapid prototyping is to construct prototypes of proposed software systems fast and with little work. Then these prototypes can be used to evaluate the design and the requirements of software systems.

The modeling strategy is important for making the rapid prototyping approach work for real-time systems. Timing constraints complicate the design of real-time systems because they introduce interactions between otherwise unrelated parts of the system. A good modeling strategy helps to counteract this effect. This can be done in the following ways [LUQ93]:

- By decoupling behavioral aspects of a system from its timing properties to allow independent analysis of these two aspects, and
- By organizing timing constraints in a hierarchical fashion, to allow independent consideration of smaller subsets of timing constraints.

An effective modeling strategy, therefore, supports a set of abstractions useful for simplifying the timing aspects of systems with hard real-time constraints. PSDL supports a modeling strategy based on dataflow graphs augmented with non-procedural timing and control constraints [LUQ93]. Figure 2.1 shows a sample PSDL graph.

Distributed systems can be generally implemented using three models: Client/Server Model, Distributed Object Model, and Tuple Space Model [KIN99].

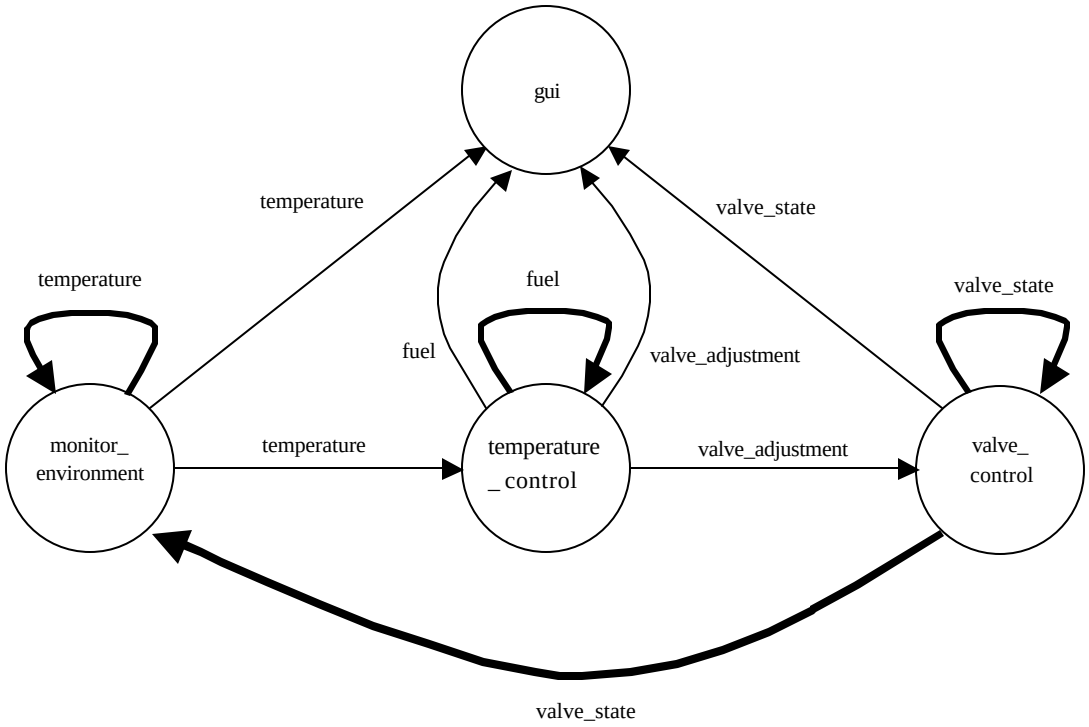

Figure 2.1 A PSDL Graph

1. Client/Server Model

The Client/Server model contains a set of server and client processes. A server process acts as a resource manager for a collection of resources of a particular type such as database server, file server, print server, etc. A client process communicates with the server for the purpose of exchanging or retrieving information. Communication between client and server can be achieved through sets of protocols agreed by both parties.

The Client/Server model provides an effective general-purpose approach to sharing of information and resources in distributed systems. It is possible for both the client and server processes to be run on the same computer. Moreover, some processes are both client and server processes [CAA98]. That is, a server process may use the services of another server, appearing as a client to the latter [SIN97].

The major drawback of the Client/Server model is that the control of individual resources is centralized at the server and this could create a potential bottleneck and a

single point of failure. Although many implementations have tried to overcome this drawback by replicating storage data and functions across multiple servers (thus making duplicate servers to either act as backups or serve different clusters of clients), this has introduced new problems in terms of maintaining data consistency in servers.

a. Sockets

Sockets are low-level inter-process communication mechanisms similar to file input/output mechanisms. These require developers to implement their own protocols through which the client and the server communicate.

b. Remote Procedure Call (RPC)

Bierrel and Nelson [BAN84] introduced a different way to approach the client-server model. They suggested that programs should be allowed to call procedures located in other machines. This approach is known as Remote Procedure Call (RPC) [CAA98].

RPC is a high-level communication paradigm that allows network applications to be developed by way of specialized procedure calls. The major limitation of RPC is that it only offers synchronous data exchange between the calling program and the called procedure. Using RPCs to integrate applications also limits portability because the application code becomes very dependent on the operating system. RPC is a widely used technique that underlies many distributed operating systems [TAN95].

c. Message Oriented Middleware (MOM)

MOM is middleware that facilitates communication between distributed applications. It supports both synchronous and asynchronous messaging. MOM sends messages from one application to another using a queue as an interim step. Client messages are sent to a queue and remain there until they are received by the server application. The advantage of this system is that the server application does not need to be available when the message is sent. MOM can also facilitate retrieval of messages

using priority and load-balancing schemes. MOM can also provide a level of fault tolerance using persistent queues.

2. Distributed Object Model

A distributed object based system isolates requestors of services from providers of services by a well-defined interface. In a distributed object model, a client sends a message to an object that in turn interprets the message to decide what service to perform. This service could be performed either through an object or a broker.

Distributed object systems such as CORBA [ORB01] [CAD97] [ORB95], DCOM [CAD97] [DCO97], and Java RMI [RMI00] provide the infrastructure for supporting remote object activation in a client transparent way. A client application obtains a pointer (or a reference) from a remote object, and invokes methods through that pointer as if the object resides in the client's own address space. The infrastructure takes care of all low-level issues such as packing the data in a standard format for heterogeneous environments, maintaining the communication endpoints for message sending and receiving, and dispatching each method invocation to the target object.

3. Tuple Space Model

Tuples are typed data structures. Collections of tuples exist in a shared repository called a "tuple space". Coordination is achieved through communication taking place in a tuple space that is globally shared among several processes. Each process can access the tuple space by inserting, reading or withdrawing tuples [LBS01].

In this model, the programmer never has to be concerned with program explicit message passing constructs and never has to manage the relatively rigid, point-to-point process topology induced by message passing. Coordination is uncoupled and anonymous. Here, "uncoupled" means the acts of sending (producing) and receiving (consuming) data are independent of each other (akin to message passing). "Anonymous" means processes' identities are unimportant and, in particular, there is no need to "hard wire" them into the code.

E. SUMMARY

In this chapter, we first discussed the benefits and challenges of distributed systems. We pointed out that performance, scalability, resource sharing, fault tolerance and availability, and elegance are the areas that distributed systems provide more benefits than standalone systems. But, we also pointed out the challenges introduced by distributed systems such as latency and synchronization. Then we discussed the modeling strategies to develop distributed systems. We emphasized the importance of rapid prototyping in development of distributed systems.

III. JINI AND JAVASPACES TECHNOLOGIES

In this chapter, we investigate Jini and JavaSpaces [EDW01] technologies in detail. We discuss the basic characteristics of these technologies and explain how we can use these technologies for inter-process communication in distributed real-time systems.

A. JINI

Jini is one of a large number of distributed system architectures (CORBA and DCOM are others). It is distinguished by being based on the Java programming language and deriving many features that leverage the capabilities that this language provides, such as object-oriented programming, code portability, RMI, network support, and security.

In general, Jini technology is used for networking embedded systems that contain a microprocessor and do a specific task. More specifically, Jini technology gives network devices self-configuration and self-management capabilities; it lets devices communicate immediately on a network without requiring human intervention [SAW01].

Jini technology consists of a programming model and a runtime infrastructure. The programming model helps designers build reliable distributed systems as a federation of services and client applications. The runtime infrastructure resides on the network and provides mechanisms for adding, subtracting, locating, and accessing services. Services use the runtime infrastructure to make them available when they join the network. Clients use the runtime infrastructure to locate and contact desired services. Once the services have been contacted, the client can use the programming model to enlist the help of services in achieving its goals. Some of the features of Jini include [EDW01]:

- Enabling users to share services and resources over the network,
- Providing users easy access to resources anywhere on the network while allowing the network location of the user to change, and

- Simplifying the task of building, maintaining, and altering a network of devices, software and users.

Jini redefines the concept of a client. Instead of providing a fixed set of “local” devices, Jini supplies the Java client with a federation of remote “plug and play” devices in a dynamic configuration (the federation) that is personalized for each client [EDW01].

Jini is focused around four main areas: Simplicity, Reliability, Scalability, and Device Genericity [EDW01].

1. Simplicity

Jini defines how services connect to one another; it does not define what those services are, what they do, or how they work. In fact, Jini services can even be written in a language other than Java; the only requirement is that there exists, somewhere on the network, software that is written in Java to participate in the mechanisms Jini uses to find other Jini devices and services. From the perspective of Jini, everything, even a device such as a scanner, printer or telephone, is really a service. To use an object-oriented metaphor, everything in the world, even hardware devices, can be understood in terms of the interfaces they present to the world. These interfaces are the services they offer, so Jini uses the term “service” explicitly to refer to some entity on the network that can be used by other Jini participants. The services these entities offer may be implemented by some hardware device or combination of devices, or some pure software component or combination of components.

2. Reliability

Jini does have similarities to a name server; it even provides a service for finding other services in a community. But there are two essential differences between what Jini does and what simple name servers do.

Jini supports serendipitous interactions among services and users of those services. That is, services can appear and disappear on a network in a very lightweight way. Interested parties can be automatically notified when the set of available services changes. Jini allows services to come and go without requiring any static configuration or administration. In this way, Jini supports what might be called "spontaneous networking". Furthermore, every device or service that connects to a Jini community carries with it all the code necessary for it to be used by any other participant in the community.

Communities of Jini services are largely self-healing. This is a key property built into Jini from the ground up. Jini does not make the assumption that networks are perfect, or that software never fails. Given time, a Jini system will repair damage to itself. Jini also supports redundant infrastructure in a very natural way, e.g. Jini lookup services on multiple redundant machines reduce the possibility that services will be unavailable if key machines crash. A Jini client that loses contact with a server can recover and continue processing [SAW01].

These properties make Jini virtually unique among commercial-grade distributed systems infrastructures. These properties ensure that a Jini community will be virtually administration-free. Spontaneous networking means that the configuration of the network can be changed without involving system administrators. The ability for a service to carry with it the code needed to use it (via dynamic class loading) means that there is no need for driver or software installation to use a service. Furthermore, the self-healing nature of Jini also reduces administrative load.

3. Scalability

Jini addresses scalability through federation. Federation is the ability for Jini communities to be linked together, or federated, into larger groups. Ideally, the size for a single Jini community is about the size of a workgroup – that is, the number of printers, PDAs, cell phones, scanners, and other devices and network services needed by a group of 10 to 100 people.

Jini supports access to other services in other communities via federating them together into larger units. Specifically, the Jini lookup service (the entity responsible for keeping track of all the services in a community) is itself a Jini service. The lookup service for a given community can register itself in other communities, essentially offering itself up as a resource for users and services there.

4. Device Genericity

Jini is generic with regard to devices. This means that Jini is designed to support a wide variety of entities that can participate in a Jini community. These "entities" may be devices or software or some combination of both; in fact, it is generally impossible for the user of one of these things to know which it is. This is one of key contributions of Jini. To use an "entity" (or service) we do not have to know whether that "entity" is hardware or software. All we need to know is the interface this entity presents.

B. JAVASPACES

JavaSpaces is a service of Jini Technology [FHA99]. It is a high-level coordination tool for gluing processes together into a distributed application. It is also a departure from conventional distributed tools, which rely on passing messages between processes or invoking methods on remote objects. JavaSpaces provides a fundamentally different programming model that views applications as a collection of processes cooperating via the flow of objects into and out of one or more spaces. This space-based model (a tuple-space model that we discussed in previous chapter) of distributed computing has its roots in the Linda [GEL85] coordination language developed by Dr. David Gelernter at Yale University. There are a few similar implementations like JavaSpaces, IBM's Tspaces [TSP00] and Cloudscape's Java database [CLO00], which are built based on tuple-space model.

A *space* is a shared, network-accessible repository for objects. Processes use the repository as persistent object storage and exchange mechanism; instead of communicating directly, they coordinate by exchanging objects through spaces.

Processes perform simple operations to *write* new objects into a space, *take* objects from a space, or *read* (make a copy of) objects in a space. Processes use a simple value-matching lookup to find the objects that matter to them in the case of reading or taking an object from the space. If a matching object is not found immediately, then a process can wait until one arrives. Unlike conventional object stores, processes do not modify objects in the space or invoke their methods directly--while there, objects are just passive data. To modify an object, a process must explicitly remove it, update it, and reinsert it into the space.

To build space-based applications, we must design *distributed data structures* and *distributed protocols* that operate over them [FHA99]. A distributed data structure is made up of multiple objects that are stored in one or more spaces. For example, an ordered list of items might be represented by a set of objects, each of which holds the value and position of a single list item. Representing data as a collection of objects in a shared space allows multiple processes to concurrently access and modify the data structure.

Distributed protocols define the way participants in an application share and modify these data structures in a coordinated way. For example, if our ordered list represents a queue of printing tasks for multiple printers, then our protocol must specify the way printers coordinate with each other to avoid duplicating efforts. Our protocol must also handle errors: otherwise a jammed printer, for example, could cause many users to wait unnecessarily for jobs to complete, even though other printers may be available. While this is a simple example, it is representative of many of the issues that crop up in more advanced distributed protocols.

Distributed protocols written using spaces have the advantage of being *loosely coupled* because processes interact indirectly through a space (and not directly with other processes). Data senders and receivers are not required to know each other's identities or even to be active at the same time. Conventional network tools require that all messages be sent to a particular process (who), on a particular machine (where), at a particular time (when). Instead, using a JavaSpaces system, we can write an object into a space with the expectation that someone, somewhere, at some time, will take the object and make use of

it according to the distributed protocol. Uncoupling senders and receivers leads to protocols that are simple, flexible, and reliable. For instance, in our printing example, we can drop printing requests into the space without specifying a particular printer or worrying about which printers are up and running, since any free printer can pick up a task.

The JavaSpaces technology's shared, persistent object store encourages the use of distributed data structures, and its loosely coupled nature simplifies the development of distributed protocols.

1. Key Features

The JavaSpaces programming interface is simple, to the point of being minimal. Applications interact with a space through a handful of operations. On the one hand, this is good -- it minimizes the number of operations you need to learn for writing real applications. On the other hand, it begs the question: how can we do such powerful things with only a few operations? The answer lies in the space itself, which provides a unique set of key features: Shareness, Persistentness, Associativeness, Transactional Secureness, and Exchange of Executable Content [EDW01].

a. Shareness

Spaces are network-accessible "shared memories" that many remote processes can interact with concurrently. A space itself handles the details of concurrent access, letting us focus on the design of our clients and the protocols between them. The "shared memory" also allows multiple processes to simultaneously build and access distributed data structures, using objects as building blocks.

b. Persistentness

Spaces provide reliable storage for objects. Once stored in the space, an object will remain there until a process explicitly removes it. Processes can also specify a

"lease" time for an object, after which it will be automatically destroyed and removed from the space.

Because objects are persistent, they may outlive the processes that created them, remaining in the space even after the processes have terminated. This property is important and necessary for supporting uncoupled protocols between processes. Persistence allows processes to communicate even if they run at non-overlapping times. For example, we can build a distributed "chat" application that stores messages as persistent objects in the space and allows processes to carry on a conversation even if they are never present at the same time (similar to email or voice mail). Object persistence can also be used to store preference information for an application between invocations -- even if the application is run from a different location on the network each time.

This feature of JavaSpaces may also introduce some problems related to the storage capabilities. JavaSpaces storage capacity places a constraint on how many objects we can store in the space at a given time. The number of objects needed to be stored in the JavaSpaces must not exceed this capacity. Our application may exceed JavaSpaces capacity by not controlling the flow and lifetime of objects stored in the JavaSpaces. Objects must be removed from the JavaSpaces when we no longer need them. Lifetime of objects stored in the JavaSpaces must be carefully designed so that we never exceed the capacity of the JavaSpaces at any time.

c. Associativeness

Objects in a space are located via *associative lookup*, rather than by memory location or by identifier. Associative lookup provides a simple means of finding the objects we are interested in according to their content, without having to know what the object is called, who has it, who created it, or where it is stored. To look up an object, we create a template (an object with some or all of its fields set to specific values, and the others left as null to act as wildcards). An object in the space matches a template if it matches the template's specified fields exactly.

d. Transactional Secureness

The JavaSpaces technology provides a transaction model that ensures that an operation on a space is atomic (either the operation is applied, or it is not). Transactions are supported for single operations on a single space, as well as multiple operations over one or more spaces (either *all* the operations are applied, or *none* are). Transactions provide a way to deal with partial failure. If an operation fails in a transaction, then all operations under this transaction fail and none of them are applied.

e. Exchange of Executable Content

While in the space, objects are just passive data -- we cannot modify them or invoke their methods. However, when we read or take an object from a space, a local copy of the object is created. Like any other local object we can modify its public fields as well as invoke its methods, even if we have never seen an object like it before. This capability gives us a powerful mechanism for extending the behavior of our applications.

2. Advantages of JavaSpaces Technologies

If our application can be modeled as a flow of objects into and out of spaces (as many can), then the JavaSpaces technology offers a number of compelling advantages over other network-based software tools and libraries. Simplicity, expressiveness, loosely coupled protocols, and code design are basic advantages of JavaSpaces [EDW01]:

a. Simplicity

The technology does not require learning a complex programming interface; it consists of a handful of simple operations.

b. Expressiveness

Using a small set of operations, we can build a large class of distributed applications without writing a lot of code.

c. Loosely Coupled Protocols

By uncoupling senders and receivers, spaces support protocols that are simple, flexible, and reliable. “Uncoupling” facilitates the composition of large applications (we can easily add components without redesigning the entire application), supports global analysis (we can examine local computation and remote coordination separately), and enhances software reuse (we can replace any component with another, as long as they abide by the same protocol).

d. Code Design

When writing a server, features such as concurrent access by multiple clients, persistent storage, and transactions are reinvented time and time again. JavaSpaces technology provides these functionalities for free; in most cases, we only need to write client code, and the rest is handled by the space itself.

3. JavaSpaces Programming Model

JavaSpaces programming model is very simple and minimal. JavaSpaces has only four types of operations.

- To write a new object to the JavaSpaces.
- To read an object from JavaSpaces.
- To take an object from JavaSpaces.
- To ask JavaSpaces notify us when objects that match a certain template are written into the space [EDW01].

The JavaSpaces interface is given in Appendix A.

a. write() method

The write() method is used to deposit a new entry into JavaSpaces. The signature of the write() method is as follows:

```
Lease write (Entry e, Transaction txn, long lease)
    throws RemoteException, TransactionException;
```

The arguments are the Entry object to be written, the Transaction associated with the write operation, and the initial requested lease duration expressed in milliseconds. The Entry that is passed to the method is unchanged; it is serialized, and a copy of it is stored in JavaSpaces. The method returns a Lease object, which can be renewed manually by the client, or can be handled by a LeaseRenewalManager or other code for renewal.

If a Transaction object is passed to the write() method, then the operation will not execute until the Transaction completes successfully. If there is no need for a Transaction, then a null transaction parameter is passed to the method.

If a call to this method returns without raising an exception, this means that the entry object has been successfully written to the space. If a RemoteException is raised, it is impossible to decide if the entry object has been successfully written or not [EDW01].

b. read() and readIfExists() Methods

The read() and readIfExists() methods use the provided template to search JavaSpaces. The template is compared against the Entry objects stored in the space according to the attribute matching rules. If a match exists, this matching Entry will be

returned, otherwise null will be returned. The signatures of the read() and readIfExists() methods are as follows:

```
Entry read (Entry tmpl, Transaction txn, long timeout)
                                        throws RemoteException,
                                               TransactionException,
                                               UnusableEntryException,
                                               InterruptedException;
Entry readIfExists (Entry tmpl, Transaction txn, long timeout)
                                        throws RemoteException,
                                               TransactionException,
                                               UnusableEntryException,
                                               InterruptedException;
```

If there are multiple matching objects in the space, there is no guarantee that the same object will be returned each time. Even if there is only one matching object, a particular JavaSpaces implementation may return *equivalent* yet *distinct* objects each time; that is, JavaSpaces may return two objects that have identical values as reported by equals(), but they may be separate objects as reported by the "==" operator.

Passing a "null" template to these methods means that any Entry in the space may be returned.

The difference between these two methods is in how they use their timeout parameters. The read() method call will return a matching Entry if it exists, or wait for the timeout period until a matching Entry appears. The readIfExists() method call will try to return a matching Entry immediately if it exists, or null otherwise. It does not wait for a matching Entry to appear.

The readIfExists() method uses its timeout parameter if a matching Entry is in a transaction. It blocks for the duration specified by the timeout parameter, if the only possible match is involved in a transaction. If the transaction "quiesces" before the timeout period, then the Entry will be returned. If the timeout elapses and there is no matching Entry available that is not involved in a transaction, then null will be returned.

The read() method also considers transactions. It will wait until a matching Entry appears, or until a matching Entry that is involved in a transaction stabilizes. If the timeout elapses before either of these occurs, then null will be returned.

The NO_WAIT constant in the JavaSpaces interface is used as a timeout value to mean that these calls should return immediately [EDW01].

c. take() and takeIfExists() Methods

The take() and takeIfExists() methods use the provided template to search JavaSpaces like read() and readIfExists() methods. The signatures of the take() and takeIfExists() methods are as follows:

```
Entry take (Entry tmpl, Transaction txn, long timeout)
                                    throws RemoteException,
                                        TransactionException,
                                        UnusableEntryException,
                                        InterruptedException;

Entry takeIfExists (Entry tmpl, Transaction txn, long timeout)
                        throws RemoteException, TransactionException,
                        UnusableEntryException, InterruptedException;
```

These two methods work just like read() and readIfExists() methods respectively. They match a template, possibly block until some timeout elapses, and then return a matching Entry or null. The difference is that read methods leave the matched Entries in the JavaSpaces; the take methods remove them from the JavaSpaces [EDW01].

d. notify() Method

The notify() method is used to register for notification of future writes of a specified entry. The signature of the notify() method is as follows:

```
EntryRegistration notify (Entry tmpl,
                          Transaction txn,
                          RemoteEventListener l,
                          long lease,
                          MarshalledObject obj)
    throws RemoteException, TransactionException;
```

This method takes an Entry as a template that will be matched against future writes to the JavaSpaces. If a new Entry is written that matches the template, an event will be sent to the listener specified in the notify method call.

In addition to the template, the method takes an optional transaction parameter, a “RemoteEventListener” to send events to, requested lease duration in milliseconds, and a “MarshalledObject” that contains a serialized object that will be returned in any events generated as a result of this registration. The call to this method returns an EventRegistration object containing the source and type of the events that will come as a result of the registration, the Lease for the registration, and the last sequence number sent for the event type.

Sun's specification for the JavaSpaces dictates that the service use *full ordering* for sequence numbers of events from the service. So if the service sends an event with sequence number 5 and then sends an event with sequence number 10, this requires that there have been 4 intervening matches of written Entry objects that resulted in events not seen by the caller, possibly because of network problems or out-of-order delivery.

JavaSpaces service does not guarantee the delivery of events to the registered clients; instead it makes a "best effort" attempt. If service catches a RemoteException while trying to send an event to the client (through the invocation of notify method of the client's listener), it will periodically try to resend events until the client's lease expires [EDW01].

e. snapshot() Method

The snapshot() method is not a core operation of the JavaSpaces. It helps to make interactions with JavaSpaces much more efficient. The signature of this method is as follows:

```
Entry snapshot (Entry e) throws RemoteException;
```

The process of serializing an object in Java can be very time consuming, especially if the object is large or has a complicated series of references within it. Any object, which is passed as a parameter to one of the methods in the JavaSpaces interface, must be serialized for transmission to the service. The snapshot() method gets an Entry and returns an Entry. The returned Entry can be used in any future calls to the same JavaSpaces that this method called on, and will avoid the repeated serializations process. Essentially, the returned Entry from the call is a "token" that identifies the original object. In cases such as writing the same object many times or using an Entry as a template to search over and over again for matching objects, it may be beneficial to avoid the cost of having to serialize the same object over and over again by using the snapshot() method.

Since the returned Entry is a specialized representation of the original entry, it is only valid as a parameter to methods on the JavaSpaces that generated it. So, it is not possible to produce a snapshot of an Entry on one JavaSpaces and use this snapshot on another.

It is also not possible to compare a snapshot to Entry objects that are returned from a JavaSpaces server. This means that all of the methods that return Entry objects return "non-snapshot" objects. Snapshots are only used as input parameters, not return values.

Finally, there is no guarantee about the snapshot of a "null" parameter. The snapshot of null depends on the specific JavaSpaces implementation [EDW01].

C. SUMMARY

In this chapter, we present Jini/JavaSpaces technology in detail. We discussed the basic characteristics of this new technology. As a result, we can say that Jini/JavaSpaces technology can be used as the complete basis for a new distributed systems programming paradigm. Our idea here is that rather than building remote communication interfaces or protocols for each new distributed application, applications can be defined in terms of the set of objects they write into JavaSpaces and the set of objects they retrieve from JavaSpaces. JavaSpaces interface would be the common API for interaction between distributed applications. In the next chapter, we will discuss how we can use Jini/JavaSpaces for inter-process communication in DCAPS.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. ISSUES RELATED TO PSDL MODEL

In this chapter, we first discuss the basic characteristics of the Prototyping System Description Language (PSDL) related to real-time system design and then explain why current PSDL implementation is not sufficient for inter-process communication in the development of distributed real-time systems. Finally, we offer a solution to improve the current implementation.

We also discuss how we can implement dataflow and sampled streams defined in PSDL using JavaSpaces for inter-process communication.

A. REAL-TIME CONSTRUCTS OF PSDL

Latency and *Minimum Calling* attributes are the advanced real-time constructs of PSDL. *Latency* is defined as an upper bound on duration of the time interval between the instant a data value is written into a stream and instant the data value becomes available for reading from this stream. *Minimum Calling* period is a lower bound on the duration of the interval between two successive write events on the stream [LUQ93].

If a user does not explicitly define latency and minimum calling period for a stream, Latency and Minimum Calling Period values are set to zero by the current PSDL model. This means that this stream has no delay and has an unbounded data rate.

Latency is an important constraint for distributed systems. The latency of a communication link between two network nodes in a distributed system affects the scheduling of tasks distributed over the network nodes. PSDL supports a modeling strategy based on dataflow graphs augmented with non-procedural timing and control constraints [LUQ93].

In addition to knowing the maximum latency between two nodes, it is important to know the minimum time required to send data between two nodes in a target distributed system. This value lets us determine when to start to listen for data. In the current PSDL model, there are no mechanisms that allow us to define this minimum amount of time for the target network.

One possible way to solve this problem is to extend the PSDL model to describe both the application design and characteristics of the target network and to use the target network connection latencies as a lower bound on latency. The latency of the communication link between two nodes must not exceed the latency of the stream, which connects these two operators, declared by the user. This constraint is required but not efficient for a real-time system. While minimum latency of a communication link is a constraint of the real world and a property of the target network, the maximum latency is a constraint declared by the user and characterizes the behavior of a real-time system. The real-time systems have strict timing constraints and a failure can cause the catastrophic results. This means that it is not enough to find a link with minimum latency smaller than the latency declared by the user, but we also need to ensure that the latency of the link will not exceed the latency declared by the user at any time.

The current implementation of the PSDL editor has the following problem concerning edge latency: it is impossible for a user to define different latency values for different streams with the same names, as illustrated in Figure 4.1.

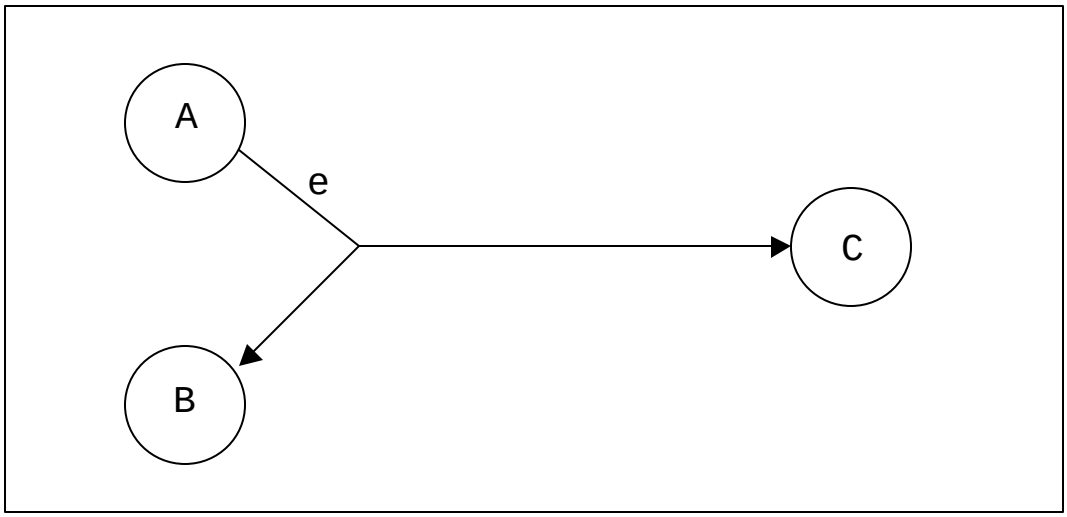

Figure 4.1 A Hyperedge

In this figure, the operator “A” is the producer and the operators “B” and “C” are the consumers. The stream has a single name (identifier) and is modeled as a single hyperedge.

The latency values between operators A-B, and A-C cannot be different because the stream is the same. This problem with the PSDL model arises because of the way hypergraphs are implemented in PSDL. PSDL expands the hyperedges as separate streams with the same name. A hyperedge may have more than one source and one destination. Figure 4.2 shows how the hyperedge in Figure 4.1 are expanded by PSDL.

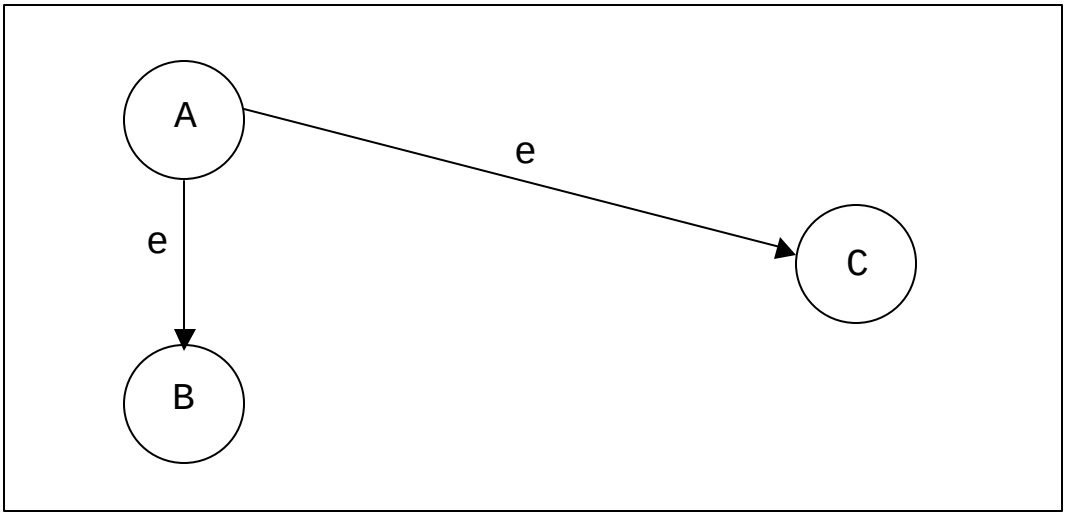

Figure 4.2 Expansion of hyperedge in Figure 4.1

For a scheduler, a stream represents the communication link between two operators. The communication link may or may not be the same even though the stream names are the same. If we do not model the scheduler this way, we lose the opportunity to distribute operators over a network efficiently. Let us assume that our scheduler tries to distribute the sample graph in Figure 4.2 as in Figure 4.3 and the latency defined by the user for the edge e is 0 ms.

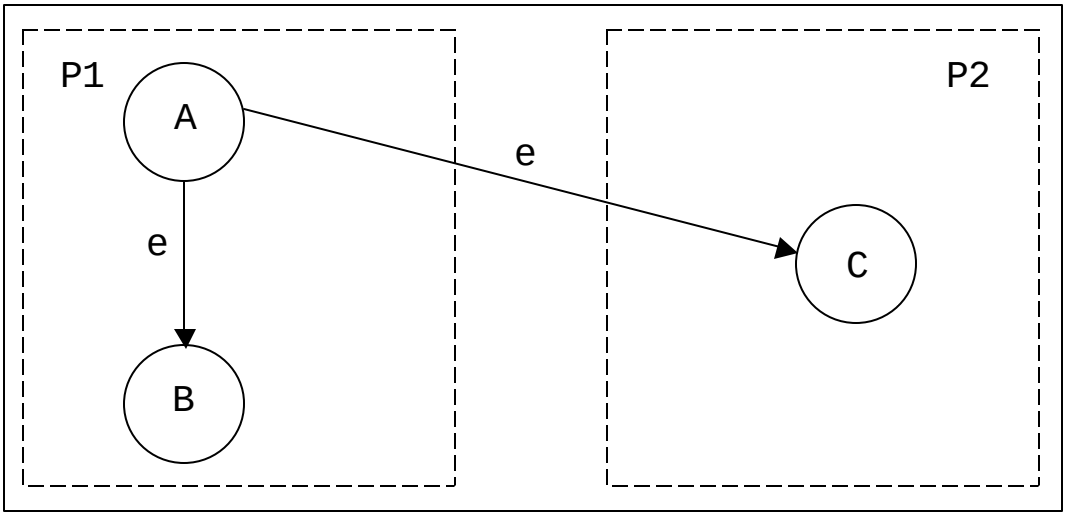

Figure 4.3 Partition of PSDL Graph over two Processors

Let us assume that the important latency is the latency between A-B and we want it to be zero. It is impossible for us to define different latency values for the stream between A-C in the current DCAPS tools because of the name of the edge connects these operators are the same with the edge connects A-B. If our scheduler tries to distribute operators as shown in Figure 4.3, and the latency of communication link between Processor 1 and 2 is 100ms then this distribution would be invalid.

Another problem with the current PSDL model is the default value for the latency. If a user does not define the latency explicitly, PSDL assumes that the latency for this link is zero, which means no delay. This is not a big problem if the PSDL model lets us explicitly define latencies for each producer/consumer tuple even though the stream names for the tuples are the same. It is a better approach if the DCAPS tools allow the user to set a default latency limit for all streams when the user chooses not to set an explicit value. If the latency of a stream is important for the user, the possibility of forgetting to set the latency value is unlikely. Additionally, if the system automatically sets the default value to zero and users do not want the latency to be zero, then they must enter a latency value for each edge they define. This can be time consuming and annoying in a rapid prototyping environment.

Future DCAPS implementations should treat each stream between producer/consumer tuples as a communication link and let the user enter different latency values. It should also allow users to set a default latency value for edges, when their latencies are not declared explicitly. We can say that the communication link latencies in the network should be the lower bound and the stream latencies should be the upper bound for the latency between two operators in a PSDL graph.

B. IMPLEMENTATION OF DATAFLOW AND SAMPLED STREAMS USING JAVASPACES

The implementation of dataflow and sampled streams is an important issue for building a distributed real-time system using PSDL. PSDL makes important assumptions about the networked environment. These assumptions include:

- There is no data loss,
- There are no breaks in the network, and
- The order of data arrival (produced by different producers) is not important for the consumer of the data, but data from the same producer are available to the consumer in the order they are produced

Much of the work in Jini (as in JavaSpaces) is designed to explicitly acknowledge these assumptions, rather than pretend they do not exist. JavaSpaces takes into account these assumptions and solves the problems related to them. As a result we can use JavaSpaces as a communication platform to build distributed real-time systems specified in PSDL.

JavaSpaces provides us a simple programming interface through a handful of operations. This interface eliminates the need for us to re-invent new communication protocols for each application. To build JavaSpaces based applications, we need to design distributed data structures and distributed protocols that operate over them. A

distributed data structure is made up of multiple objects that are stored in one or more spaces. Representing data as a collection of objects in a shared space allows multiple processes to concurrently access and modify the data structure. Distributed protocols define the way participants in an application share and modify these data structures in a coordinated way.

In this case, the data for the JavaSpaces would be the data represented by the stream in PSDL. The distributed protocol would describe the way to write and read this data to and from dataflow streams or sampled streams. If we want to use JavaSpaces for inter-process communication in PSDL, all we need to do is to define the distributed protocols for dataflow and sampled streams.

A PSDL data stream carries instances of an abstract data type associated with the stream, which can be a special pre-defined type representing exceptions. A dataflow stream is a discrete sequence of values, each of which has an independent meaning. The values in a dataflow stream must be transmitted in FIFO order and must not be lost or replicated. A sampled stream represents a continuous source of data, for which only the most recent value is meaningful. The most recently written value in a sampled stream must be available at all times and may be read many times or overwritten by more recent data before it is read without effecting the collective meaning of the stream [LUQ93].

These definitions of the dataflow and sampled streams are enough for us to build distributed protocols to implement these streams in JavaSpaces.

First, both streams carry instances of an abstract data type. These abstract data types can (should) be implemented as Entry objects of the JavaSpaces.

Second, we must use operations of JavaSpaces such as take(), read(), write() and notify(), to create the distributed protocols that implement (mimic) the operations of dataflow and sampled streams.

1. Dataflow Streams

In this section, we present the implementation of dataflow streams using JavaSpaces in detail. We will use NetworkDoubleFIFOBuffer as an example to explain the implementation. The primitive "double value" used by the NetworkDoubleFIFOBuffer is a 64-bit 754 floating point value [NAK00]. The implementation of NetworkDoubleFIFOBuffer is given in Appendix E Section B.4.

In PSDL, a dataflow stream is instantiated only when an operator has a control constraint "triggered by all …" associated with it. Dataflow streams can only be used if the execution rate of the producer is less or equal to that of the consumer operators. It is impossible for an operator to read from an empty data buffer because of this control constraint. Dataflow streams act as FIFO buffers and model discrete transactions.

The NetworkDoubleFIFOBuffer class is instantiated by passing the identity of the edge as a String object, the reference of the JavaSpaces that it will use and the latency of this dataflow communication link as a primitive "long type" (64 bit signed two's complement integer [NAK00]).

The distributed protocol, which is used by the NetworkDoubleFIFOBuffer, is very simple. The use of this buffer differs according to the source and destination operators of the edge that uses it. The source side application creates this buffer to write to it while the destination side application creates it to read from it. There exist two instances of the buffer for each stream. The buffer serves as a proxy (see Figure 4.4 for details).

The source side application simply creates a NetworkDoubleFIFOBuffer and uses its write() method to write to the stream. NetworkDoubleFIFOBuffer instantiates its private variables and uses write() method of the given JavaSpaces to write the double values to the stream when it is created at the source side. It creates EntryDouble objects with the given double value and sets the identity of these Entry objects to the identity of the stream.

The destination side application also creates a NetworkDoubleFIFOBuffer with the same identity of the source application used. The difference is that it sets the

notification of the buffer. The NetworkDoubleFIFOBuffer registers for the EntryDouble objects with the given stream identity when it is set. When it gets a notification from the space, it takes all the EntryDouble objects from the space until it gets the last one. It stores the double values of these Entry objects in its variable vector. When the destination application calls its read method, it returns the first double value in the variable vector.

2. Sampled Streams

In this section, we present the implementation of sampled streams using JavaSpaces in detail. We will use NetworkDoubleSampledBuffer class as an example to explain the implementation. The primitive "double value" used by the NetworkDoubleSampledBuffer is a 64-bit 754 floating point value [NAK00]. The implementation of NetworkDoubleSampledBuffer is given in Appendix E Section A.4.

If a PSDL stream has a "triggered by some" control constraint or has no control constraint, it is instantiated as a sampled stream. Sampled streams act as atomic memory cells and connect operators firing at uncoordinated rates. They model continuous data sources. It is also important for a sampled stream to assure that data are always available.

The NetworkDoubleSampledBuffer class is very similar to the NetworkDoubleFIFOBuffer class. It is also instantiated by passing the identity of the edge as a String object, the reference of the JavaSpaces, which it will use, and the latency of this dataflow communication link as a primitive "long type" (64 bit signed two's complement integer [NAK00]).

Like the NetworkDoubleFIFOBuffer, the use of this buffer differs according to the source and destination operators of the edge, which use it. The source side application creates this buffer to write to it while the destination side application creates it to read from it. There exist two instances of the buffer for each stream. The buffer serves as a proxy (see Figure 4.4 for details).

The source side application simply creates a NetworkDoubleSampledBuffer and uses its write() method to write to the stream. The NetworkDoubleSampledBuffer instantiates its private variables and uses the write() method of the JavaSpaces to write

the double values to the stream when it is created at the source side (just like the NetworkDoubleFIFOBuffer). It creates EntryDouble objects with the given double value and sets the identity of these Entry objects to the identity of the stream.

The destination side application also creates a NetworkDoubleSampledBuffer with the same identity of the source application used and sets the notification of the buffer. The NetworkDoubleSampledBuffer registers for the EntryDouble objects with the given stream identity when it is set. When it gets a notification from the space, it takes all the EntryDouble objects from the space until it gets the last one. The main difference in the distributed protocol used by the NetworkDoubleFIFOBuffer and the NetworkDoubleSampledBuffer is that the NetworkDoubleSampledBuffer only stores the double value of the last Entry object taken from the JavaSpaces in its EntryDouble attribute. When the destination application calls its read() method, it returns the double value of the EntryDouble attribute.

3. State Streams

If a network stream is specified as a state stream, then we can use a network sampled buffer to implement this state stream. Network sampled buffers have two constructors. The first constructor creates a network sampled buffer. The second constructor takes an additional parameter, which is the initial value of the data carried by the stream and creates a network buffer for this data. The second constructor is used to create a network state stream.

4. Results

There are two benefits of implementing dataflow stream and sampled stream buffers as proxies. First, they do not block the applications when they try to take an Entry object from JavaSpaces. If we try to implement these buffers as Entry objects, then each time when we try to write or read from the buffer we need to take it from JavaSpaces. If another application tries to use this buffer at the same time, then it needs to wait for us to finish our work and then write the buffer entry back to the space again. The same

problem also exists if we try to create a distributed protocol by trying to find a sequence for the method calls of JavaSpaces. Let us say we want to implement a sampled stream by ordering the method calls. For example, for the source application, first take the entry from the space then write the new one. For the destination application, just read from the space. In this scenario, if there is more than one source for the given stream, then these sources must wait for each other to take the object and write it back again to the space.

A second benefit of the proxy implementation of the network buffers is that when we try to read from the buffers, we do not need to go to the network. They simply return local variables. They store the current value of the stream as a local variable and they return this value to the caller.

Figure 4.4 shows how network buffers work. This implementation is applicable to both dataflow and sampled streams. As shown in Figure 4.4, a network buffer has two instances as proxies in the source and destination sides. Source side buffer proxy writes the given data to the JavaSpaces. When data (registered for notification) is written to the JavaSpaces, JavaSpaces sends a notification to the destination side buffer proxy. The destination side buffer proxy takes the data from the space and stores it in the local storage. If destination side application wants to read from the buffer proxy, then buffer proxy returns the data in the local storage. Because of this property, it takes a much shorter time to read from the network buffers because they act as if they are local buffers.

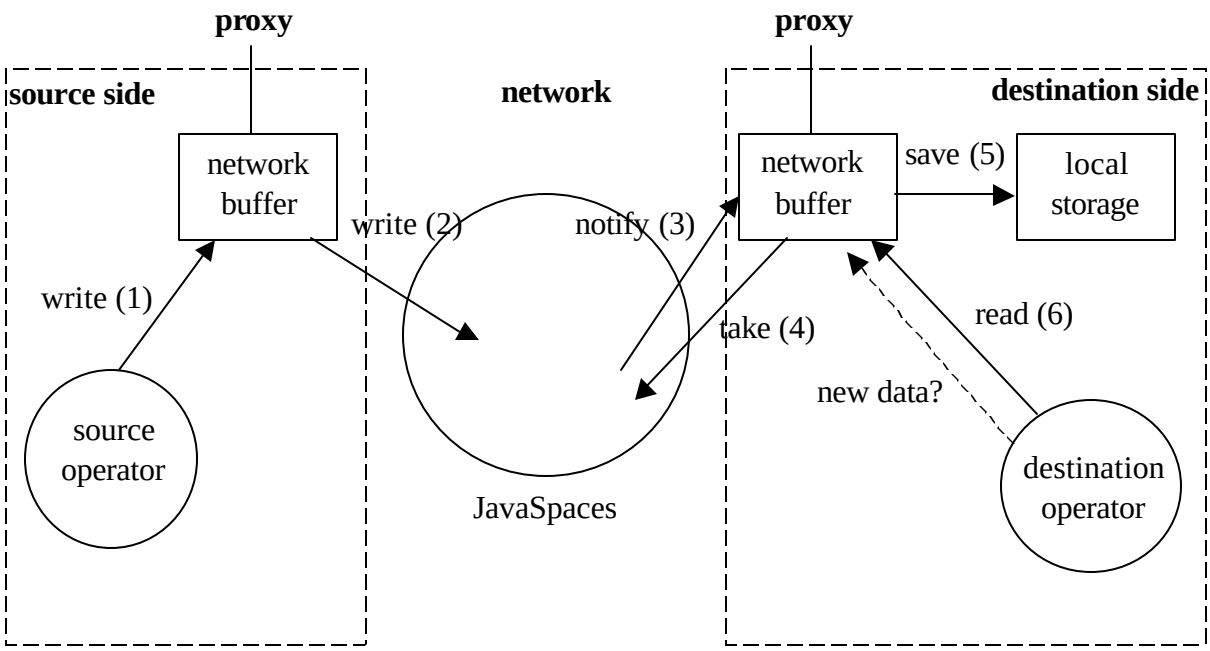

Figure 4.4 Network Buffer

Finally, both implementations create a snapshot of the entry templates so that they prevent time-consuming serialization of the Entries each time they are searched.

THIS PAGE INTENTIONALLY LEFT BLANK

V. IMPLEMENTATION

This chapter presents an example of implementing a user defined distributed real-time system PSDL specification using network buffers for inter-process communication. The purpose of this implementation is to demonstrate the use of proxy implementation of network buffers and to create a program structure that can be used to generate code from formal specifications.

A. PSDL SPECIFICATION OF THE DISTRIBUTED REAL-TIME SYSTEM

The Temperature Control System (TCS) is a good example to demonstrate the concept of network delay. Assume that the system is intended to execute in a distributed environment, consisting of two computing units. The architecture description of the TCS in PSDL is shown in Figure 5.1. The TCS consists of monitor_environment, temperature_control, valve_control and gui operators. The TCS behavior is modeled using control and timing constraint structures of the PSDL like “triggered by some” and “latency”.

The TCS controls the temperature of an environment. The environment temperature tends to increase in time. The monitor_environment operator monitors the temperature and passes this information to the temperature_control and gui operators. Temperature_control operator checks the temperature if it is between 70 – 80 degrees. If the temperature is above or below these limits, then temperature_control computes the required adjustment for the control valve and passes this information to the valve_control and gui. Valve_control reads the adjustment value and computes the new state of the valve. Valve_control passes the valve state information to monitor_environment and gui.

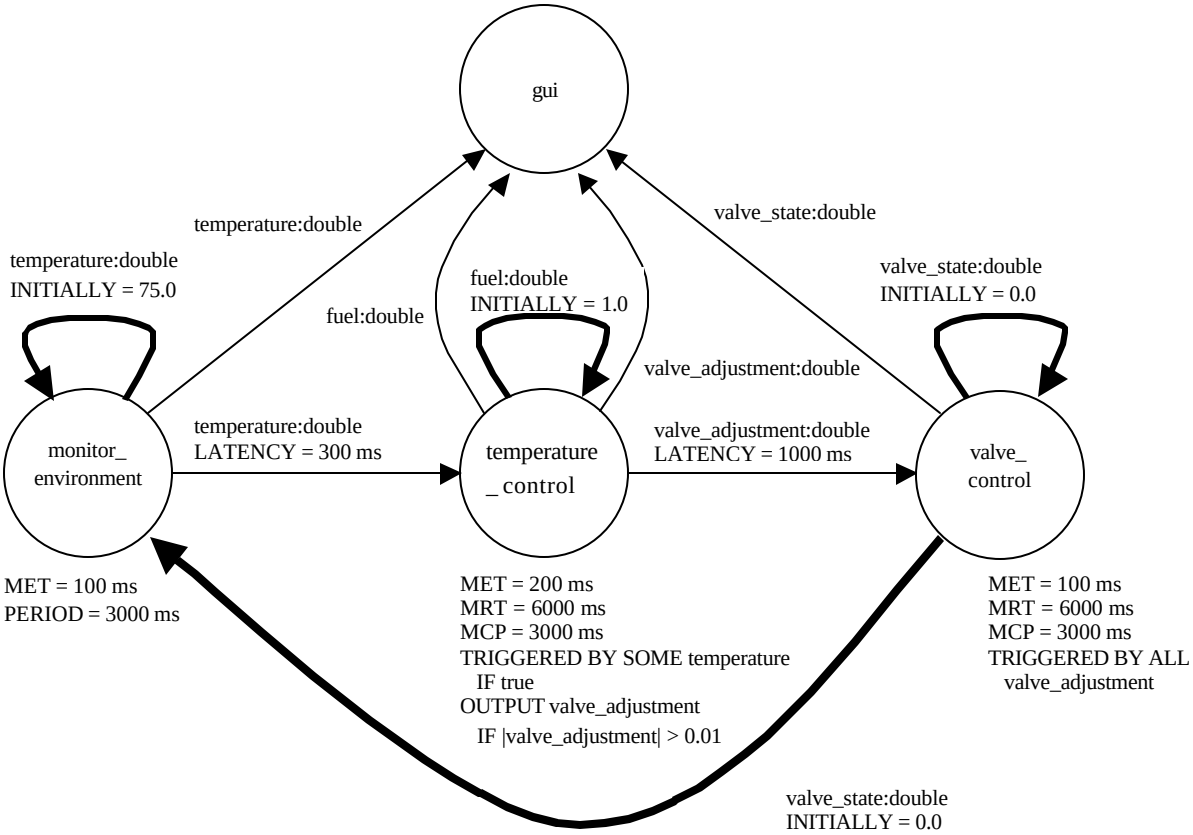

Figure 5.1 PSDL Architecture Description of Temperature Control System

The gui operator is used to present the state of the system to the user. It reports the state of the system by showing current values of the temperature, valve adjustment, and valve state. It also shows the remaining fuel and JavaSpaces status.

The temperature and valve_state state streams are used to simulate data from the target system.

B. NETWORK PARTITION

We assumed that the TCS was intended to execute on a distributed system. We also assume that the distributed system has two computation units. We need to partition the PSDL specification of the TCS to deliver the operators over these computation units.

We must take into account the timing constraints to be able to partition the TCS efficiently. Figure 5.2 shows a possible partition of the TCS. This partition places the monitor_environment and temperature_control operators on the same computation unit and gui and valve_control operators on the other computation unit.

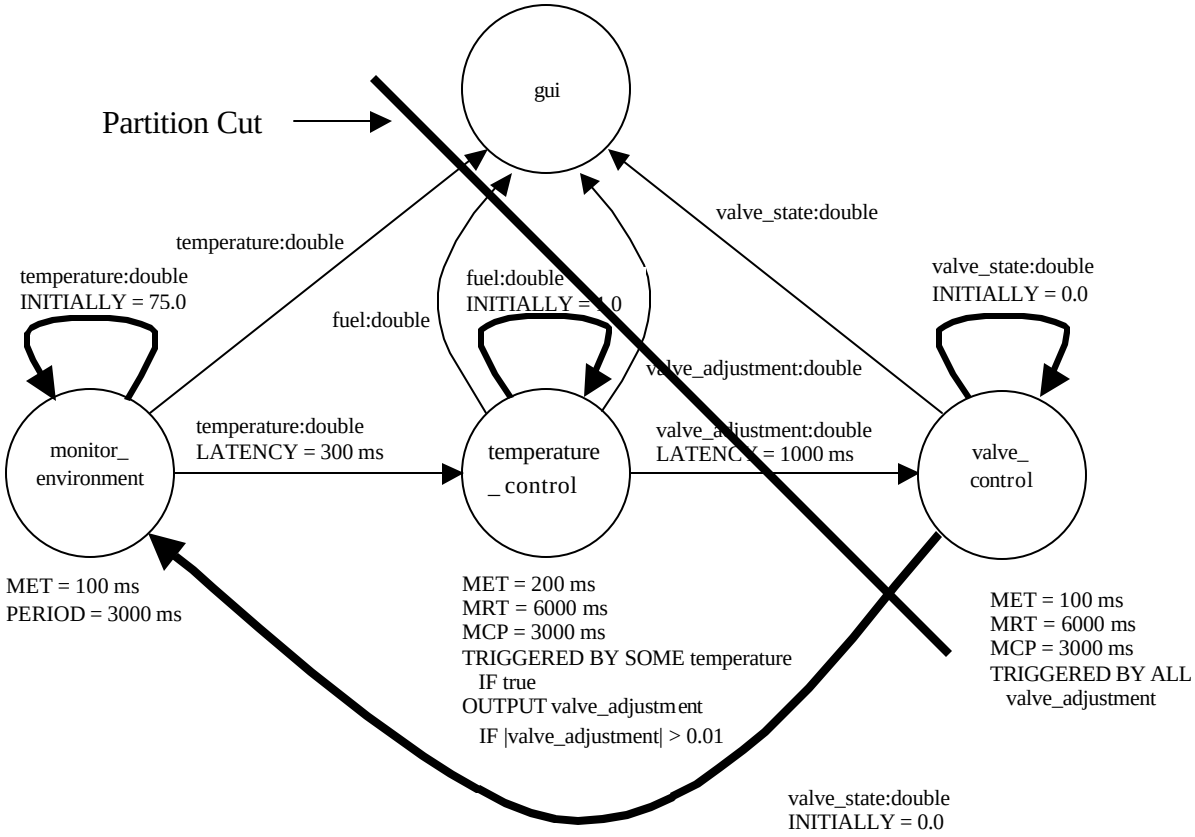

Figure 5.2 A Possible Partition of the TCS

The partition does not suggest which computation unit these operators should be delivered. The partition only suggests which operators will be executed on the same computation unit. This means that we are free to choose the particular computation unit if there are no other constraints. There can be a user-defined constraint that requires a specific operator to be executed on a specific computation unit. Additionally, the capacity of a specific computation unit may not be sufficient to execute a specific operator

presenting a constraint that must be satisfied. In our implementation, we choose the computation unit 1 to execute the gui and valve_control operators and computation unit 2 to execute the monitor_environment and temperature_control operators. We assume that all of the operators can be delivered to either of these computation units.

Let us analyze this partition to determine whether or not it is feasible.

The temperature_control and valve_control operators are sporadic operators. We can find their corresponding "triggering period" using:

$$MET \quad TP \quad \min(MRT\text{-}MET, MCP)$$

and their "finish within" values using:

$$FW = \min(TP, MRT\text{-}TP)$$

According to these formulas, "triggering period" and "finish within" values for the temperature_control operator are 3000 ms and 3000 ms respectively. "Triggering period" and "finish within" values for the valve_control operator can be found as 3000 ms and 3000 ms respectively.

We can now calculate the Least Common Multiple (LCM) of the TCS. We have three time critical operators. All of them have the same period value of 3000 ms. Therefore, the LCM of the TCS is 3000 ms. Figure 5.3 shows the timing analysis of the TCS using the partition in Figure 5.2.

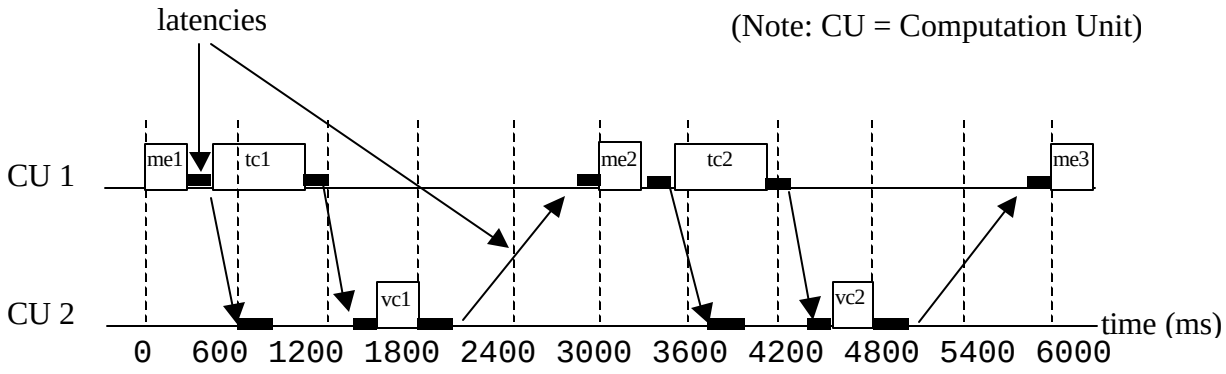

Figure 5.3 Timing Analysis of TCS

As shown in Figure 5.3, it is possible to find a feasible schedule for this partition. An important point here is we need take into account the latency, which is not declared as a constraint in the specification. The valve_state state stream between valve_control and monitor_environment in the specification becomes a network state stream according to our partition as shown in Figure 5.3. As we explained in Chapter IV, if a user does not declare a latency value for an edge, we must assign a default value for it. We assume that the default latency value for the TCS is 1100 ms.

The various properties extracted from the architecture description of TCS in Figure 5.1 are summarized in Table 5.1. This table shows the source and destination operators of each stream, the type of the stream (dataflow, sampled, or state), and how we will implement the TCS streams according to their specifications (network or local). The table also shows the maximum latencies declared for each stream. If there is no latency constraint for a stream, then we assume that the latency constraint for this stream is the default latency value.

Stream Name	Source Operator	Destination Operator	Stream Type	Buffer Type	Max. Latency
temperature	monitor_environment	temperature_control	sampled	local	300 ms
temperature	monitor_environment	gui	sampled	network	1100 ms
temperature	monitor_environment	monitor_environment	state	local	1100 ms
valve_adjustment	temperature_control	valve_control	dataflow	network	1000 ms
valve_adjustment	temperature_control	gui	sampled	network	1100 ms
fuel	temperature_control	gui	sampled	network	1100 ms
fuel	temperature_control	temperature_control	state	local	1100 ms
valve_state	valve_control	gui	sampled	local	1100 ms
valve_state	valve_control	monitor_environment	state	network	1100 ms
valve_state	valve_control	valve_control	state	local	1100 ms

Table 5.1 Properties of TCS Streams

As shown in Figure 5.3, total latency is made up of two components: latency in the local computation unit and latency in the network. The reason for this is that when our application tries to write to a network buffer, it uses its own CPU to accomplish this task. The write operation is a blocking operation. We can fire a new thread to do this job but then we cannot be sure about the sequential execution of the application because of operating system issues (such as scheduling tasks). Even if we use a separate thread we cannot save CPU time because the write operation must be executed on that CPU. As a result, we must take into account the latencies introduced by the network operations in our scheduling as shown in Figure 5.3.

The GUI operator is a non-time-critical operator. Some of the streams, which are related to this operator, become network streams as a result of the partition. These streams have no defined latencies as shown in Figure 5.1 and Table 5.1. We will also use the default latency values for these streams and try to fire the gui operator in

computational unit 2 when it is available. Figures 5.4a and 5.4b show screen shots of the GUI operator used in the implementation of the TCS.

Figure 5.4a Application Screen Shot while waiting for Start Notification

Figure 5.4b Application Screen Shot while Running

C. JAVASPACES INTERFACE

When we partition our specification, the streams (with their source and destination operators) connecting different computation units become network streams. The data values, which are carried on these streams, must be passed to the destination operators using a network buffer which is different than a local buffer. It takes relatively more time to access network stream data than local data.

In Chapter IV, we defined network buffer implementations and explained their usage. We will use that implementation of network buffers in our sample implementation of the TCS.

The network buffers have the same interface as local buffers. An important feature of the network buffers implementation using JavaSpaces is that they store their data on the local machine. This feature allows the network buffers to act as if they are local buffers. Thus, using a JavaSpaces implementation, the read method access time for a network buffer is no different than that of a local buffer. Of course, the network introduces latencies for these buffers and if no data value is available for update of the buffer, users may read null values or old values of a stream. This feature of the network buffers helps us to implement them in separate threads so that our main application does not have to wait for a network operation. Network buffers register themselves to JavaSpaces and then start to wait for the event (data) notifications from JavaSpaces. The application may use its main thread to invoke the required methods of these buffers when it gets those notifications from JavaSpaces or it may create a new thread to invoke the required methods. In each case, our application creates a simple thread for the execution of the harmonic block separated from the other threads of execution.

D. PROGRAM STRUCTURE

We will describe the program structure of the TCS implementation in this section. The class diagram of the application part for the processor 1 is shown in Figure 5.5.

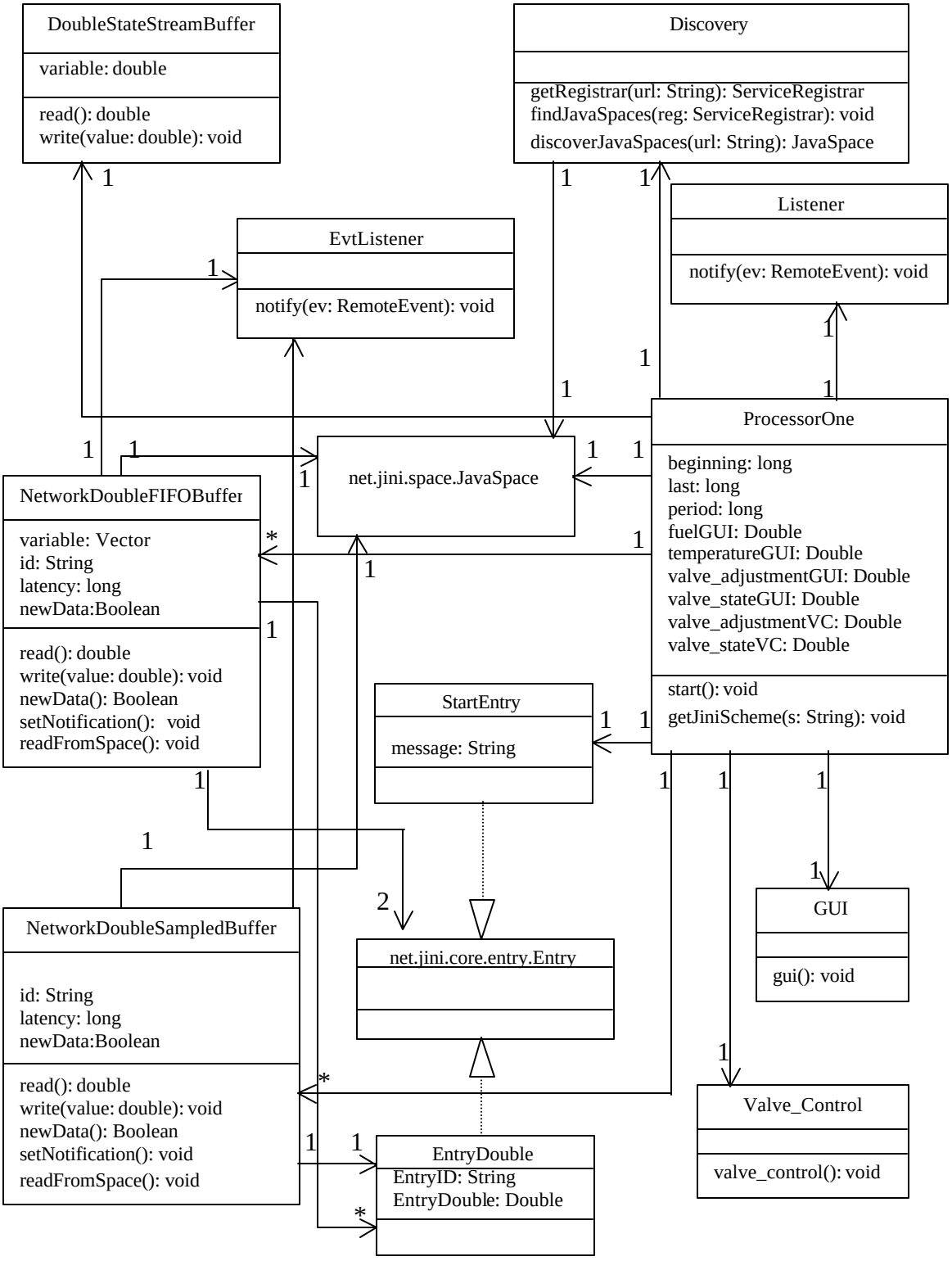

Figure 5.5 Class Diagram of Application Part 1

The program for the implementation of the TCS starts with the declaration of import statements required for source code of the local buffers, network buffers, JavaSpaces, Jini packages, SpaceDiscovery classes, RMI classes, and “swing” and “awt” classes.

Each part of the distributed application, which will execute on a different computation unit, is created as a public class. Each class has a private JavaSpaces attribute to use in the construction of network buffers as shown in Figure 5.5. The main class constructor initializes the private JavaSpaces object using the Discovery static class. The main class declares local, and network stream buffers as private attributes. Some of the other attributes of the main class are the instances of the operator objects, which are implemented as inner classes within the main class.

As shown in Figure 5.5, the main class also has other attributes such as the LCM of the application and instance of stream values. The last attributes of the main class are the “beginning” and “last” long primitive types to be used as timestamps for the harmonic block. All attributes of the main class are declared as private types and all of them are initialized in the constructor.

The operators are implemented as inner classes within the main class. The main class also has another inner class to be used as a listener for the JavaSpaces to get notifications of each period cycle.

The first inner class is the “Listener” inner class. The Listener inner class is registered and gets notifications of each period cycle from the JavaSpaces. The inner class has a default constructor, which does nothing. The reason for a default constructor for this inner class is that it throws RemoteException. Because of this we must declare this constructor explicitly. The Listener inner class has only one method called “notify()” which returns “void”. This method is called by JavaSpaces when the instances of the registered objects of this class are written to JavaSpaces. Our application calls the start() method to implement a harmonic block. The call for the start() method is placed into this notify() method.

The other inner classes are implementation specific and they contain the code for the operators. These inner classes declare a method with the same name of the operator in the specification of the application. These methods are called to fire the operators by the "start" method of the main class. These operator inner classes have their own attributes to be used as stream variables for the operator itself.

The most important part of the main class is its constructor. The constructor starts with the initialization of the JavaSpaces attribute. The constructor uses the discoverJavaSpace() method of the static Discovery class. If the initialization fails (the discoverJavaSpace() method returns null) the application informs the user and exits with error code 0.

If our application manages to find and gets a reference to the JavaSpaces object, then it starts the initialization of the other attributes. It first starts with the initialization of local, and network buffers. Network buffers need a reference to a JavaSpaces object so our application passes its own JavaSpaces object reference to the constructors of these network buffers.

When our application finishes with the initialization of the buffers, it starts the initialization of operator instances that were implemented as inner classes.

The last thing that occurs in the constructor is the registration of the objects for period start time. Our application constructor creates a new instance of a StartEntry object, and a Listener object to pass to the notify() method of the JavaSpaces object. It then informs the user by printing the statement "Waiting for start notification…" to the standard output path. The notify() method of the JavaSpaces interface may throw RemoteException or TransactionException as explained in Chapter IV. If this happens our application informs the user using the standard output path and exits with error code 0.

The last point about the application is its start method. The Listener inner class's notify() method calls this method each time it gets a notification from JavaSpaces. This method creates a new thread and fires the operators by calling their corresponding methods according to the timing constraints defined for this application. The start method

reads the required stream variables from the buffers and does the required checks for the control constraints related to each operator.

The application we explained here is only one part of our implementation of the TCS for a two node distributed application. The other part of the implementation, which will be executed on the other computation unit, has the same characteristics except that it initializes its own local, network and state streams. The other part will also have different inner classes, which implement operators to be executed on the second computation unit.

E. MASTER APPLICATION

We need to control our distributed application by using a global clock to synchronize the execution of each program unit. We stated that our application starts to wait after initialization of environment variables. All we need to do is to notify our applications to start execution. Additionally, we must notify them at the beginning of each period cycle to synchronize them.

One way to undertake this synchronization is to use a master application. This master application must start the distributed application when our distributed application parts are ready to run.

The Starter class is the master application for our TCS implementation. Figure 5.6 illustrates a screen shot of this master application. The Starter master application asks for the JavaSpaces Codebase property, a system policy file, and the LCM of the distributed application. The default values are printed for us by getting the required information from the system properties (if those properties are set by passing this information to the master application in the command line).

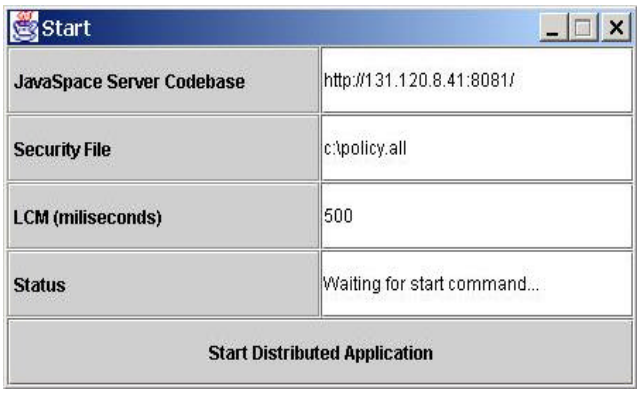

Figure 5.6 Master Application Screen Shot

Upon execution of the "Start Distributed Application" button, the Starter master application creates a new instance of StartEntry and writes it to the JavaSpaces. The Starter master application writes this entry in the beginning of each LCM (which is given to the Starter master application by the user) cycle to synchronize the distributed application.

F. RESULTS

The purpose of this implementation was to identify the efficiency of the proxy implementation of the network buffers and to create a program structure to generate code. We ran this application on a local area network that consisted of a Windows NT machine and Unix Workstation machine, isolated from the rest of the network in order to decrease the network traffic. We wanted to observe if our application was capable of properly running in a heterogeneous environment. We used the WindowsNT machine as computation unit 1 and the Unix Workstation as computation unit 2 in the first experiment and then switched the machines for the second experiment (Windows NT as computation unit 2 and Unix Workstation as computation unit 1). The JavaSpaces server was running on the WindowsNT machine. The results of the implementation showed that our proxy implementation of network buffers are efficient and properly handle the inter-process communication but unpredictable latencies introduced by the network sometimes caused our application fail. We observed that proxy implementation of the network

buffers work properly for inter-process communication but the underlying network must have real-time support. Another important point is that we ran the JavaSpaces server on the WindowsNT machine. This increased the load of computation unit 1 and increased the latencies for JavaSpaces operations.

VI. EXPERIMENTS AND RESULTS

This chapter presents the results of experiments conducted to measure the response time of Jini/JavaSpaces service for inter-process communications. The behavior of a network is unpredictable because of the variable queuing delays, congestion losses, routing protocols, and traffic load. A distributed real-time application has strict requirements for throughput, delay, jitter, and packet loss. These requirements are difficult to meet in an environment with variable queuing delays and congestion losses. For an IP-based network that provides only a best-effort service, the tools for controlling congestion and providing service are limited [STA99]. Also, the response time of the Jini/JavaSpaces service not only depends on the underlying network but also depends on the computing power of the machine we run these services. Because of these reasons, it is hard to determine a specific response time of Jini/JavaSpaces service. Instead, we tried to determine an average response time against which we can compare the Jini/JavaSpaces service performance.

Name	OS Name	Version	System Type	Processor	Total Phys. Mem.	Total Virt. Mem.	Network Connection
Turtle1	Microsoft Windows 2000 Pro	5.0.2195 Service Pack 1 Build 2195	X86 based PC	Pentium 4 X86 Family 15 Model 0 Stepping 7 ~1396 Mhz	524 KB	1800 MB	10/100 Mbps
Sun58	Solaris	Release 5.7	SUN workstation	Sparc 10 40 Mhz	96 MB	1056 MB	10 Mbps
Norma	Solaris	Release 5.7	SUN workstation	UltraSparc11i 270 Mhz	128 MB	1107 MB	100 Mbps
Saturn	Solaris	Release 5.7	SUN workstation	UltraSparc11i 270 Mhz	128 MB	1107 MB	100 Mbps
Moon	Solaris	Release 5.7	SUN workstation	UltraSparc11i 300 Mhz	128 MB	1107 MB	100 Mbps

Table 6.1 Systems used in Experiments

As mentioned, DCAPS targets heterogeneous distributed system development. As a result of this, we need to test the performance of the Jini/JavaSpaces service in a heterogeneous environment. We used different operating systems and machines to test the performance of Jini/JavaSpaces for this purpose. Table 6.1 shows the specifications of the systems we used for our experiments.

A. TEST PROGRAM

A simple test program was developed and used for the experiments. Figures 6.1a, b, and c show screen shots of this test program.

Figure 6.1a Test Program Main Window

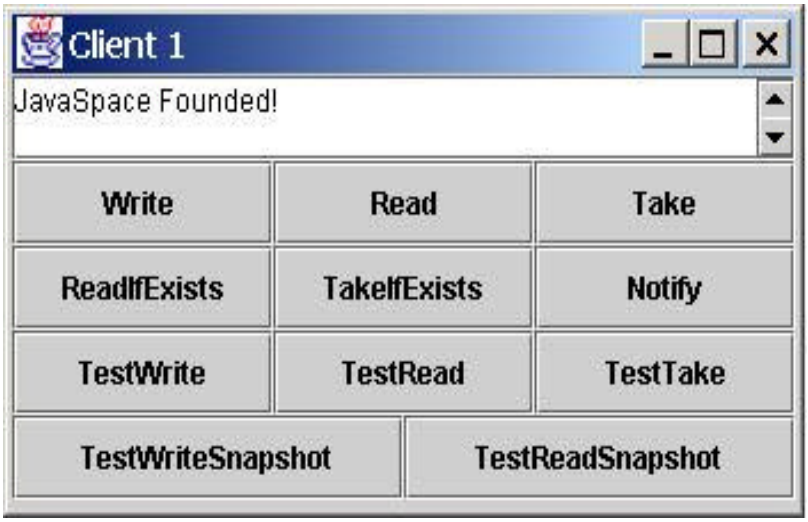

Figure 6.1b A Client created by Test Program

Figure 6.1c A Pop-up Dialog used by Test Program

The Sun58 workstation and Turtle1 resided on the same local area network. These two machines were isolated from the other machines (Norma, Saturn, Moon) in the LAN by a bridge to reduce the network traffic for experiment purpose.

The test program generates clients for the Jini/JavaSpaces service. These clients are capable of discovering a local Jini/JavaSpaces service or can be directed to discover a Jini/JavaSpaces service, which is not local, by passing the IP address of the Jini lookup service.

The clients generated by the test program have a simple GUI that allows users to use services of JavaSpaces. The clients also allow us to try different experiments. These experiments are: test for write() method, test for read() method, test for take() method, and test for write()/read() methods using snapshot() method. These default experiments ask users to enter desired experiment specific values such as number of attempts for write() method. All experiments log the results of the experiments to a file passed to the test program as a command line argument. We developed different experiment scenarios by combining default experiments or by generating several clients on the same or different machines.

B. EXPERIMENTS

The response time diagrams of the JavaSpaces server for different services are illustrated in this section. We conducted different experiments to measure an average response time of JavaSpaces. Experiment results are shown by a response time diagram (JavaSpaces response time for a request) for each machine (if more than one machine was used for the experiment). Each experiment uses a different scenario to test the performance of the JavaSpaces service under different conditions and determines the overall performance of the JavaSpaces service. We used the same entry for all experiments. The entry, used in the experiments, had an identity (1 byte), message (3 bytes), and a timeout value (primitive long type, 64-bit signed two's complement integer [NAK01]). Therefore, the total size of the entry was 12 bytes. The goal of the experiments was to measure an average response time of the JavaSpaces service, not to measure an average response time of the JavaSpaces service for different sizes of data.

Data statistics collected from the experiments are presented and summarized at the end of this chapter.

1. Experiment 1

The goal of this experiment was to measure the average response time of the write service of JavaSpaces under light load. We used only one client and made sure that there were no other applications running on the server side and client side (except the system tasks).

A client, created by the test program on the Sun58 workstation, contacted the JavaSpaces server, which was running on the Turtle1. The client attempted to write an entry 500 times to JavaSpaces. There were no other applications running on Turtle1 (except the JavaSpaces server and other system tasks).

The chart in Figure 6.2 shows the response times of the JavaSpaces server for each write attempt.

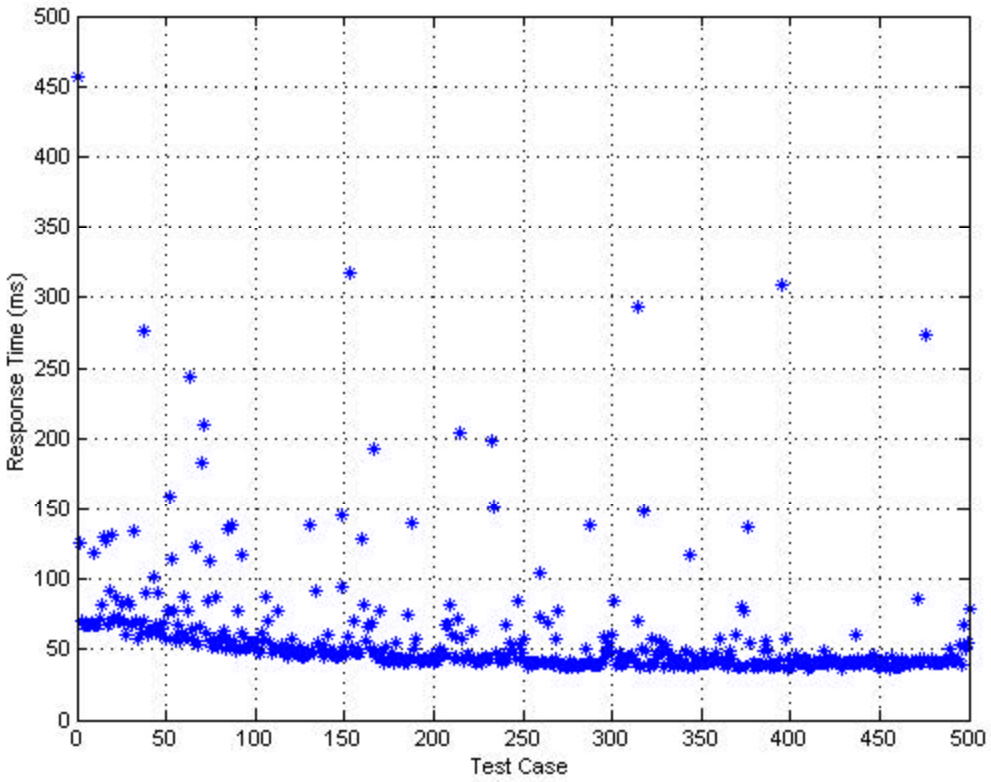

Figure 6.2 Response Time Diagram of Experiment 1

As shown in Figure 6.2, the response times were generally around 50 ms. As we expected, we had some longer response times as a result of unpredictable nature of underlying network, operating system and overhead introduced by JavaSpaces itself. As mentioned, the Turtle1 and Sun58 workstations were isolated from the rest of the network to reduce the network effects, so we conjectured that the underlying network had the minimum effect on these higher response times than the operating system and overhead introduced by JavaSpaces. An important observation was the maximum response time. The maximum response time (456 ms) occurred in the first attempt. JavaSpaces logs all contacts in a log file, so first attempt took relatively longer to process than the average response time as a result of this registration process.

2. Experiment 2

In this experiment, we tried to measure the performance of JavaSpaces under relatively higher load than the load in experiment 1. Our goal was to observe JavaSpaces performance when two different clients tried to use the write service at the same time.

Two clients, created by the test program on the Sun58 workstation, contacted the JavaSpaces server, which was running on Turtle1. Both clients attempted to write an entry 500 times to JavaSpaces at the same time. There were no other applications running on Turtle1 and Sun58 (except the JavaSpaces server and other system tasks).

The charts in Figure 6.3a and b show the response times of the JavaSpaces server for both clients. As mentioned in experiment 1, we had relatively higher response times at first attempts for both clients.

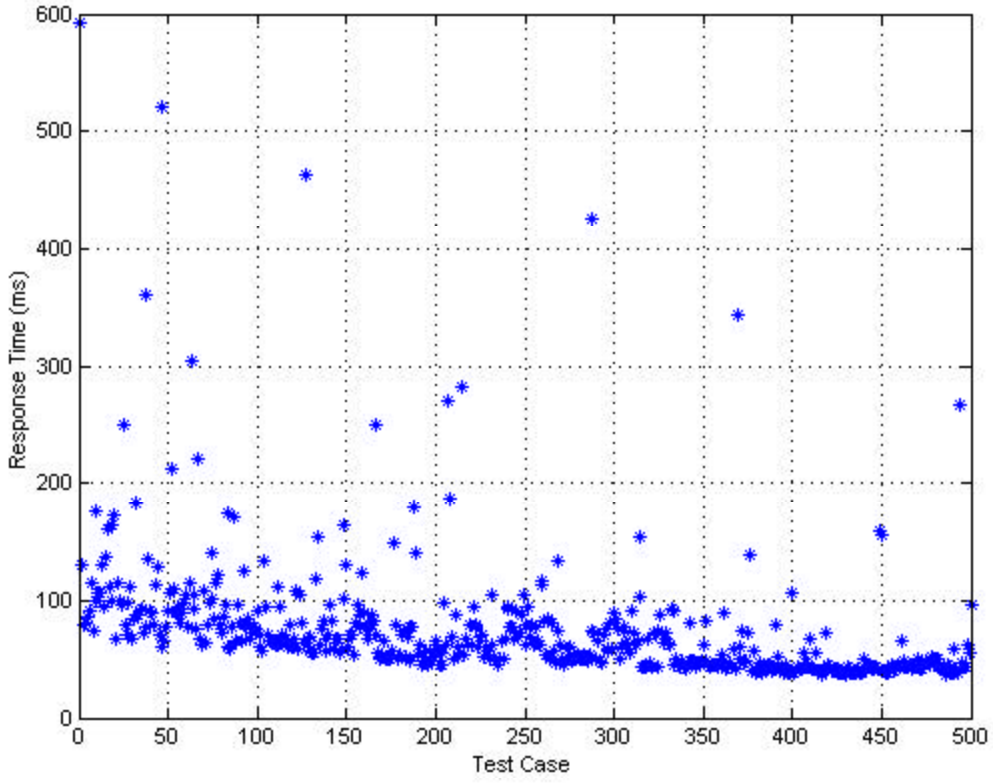

Figure 6.3a Response Time Diagram of first Client in Experiment 2

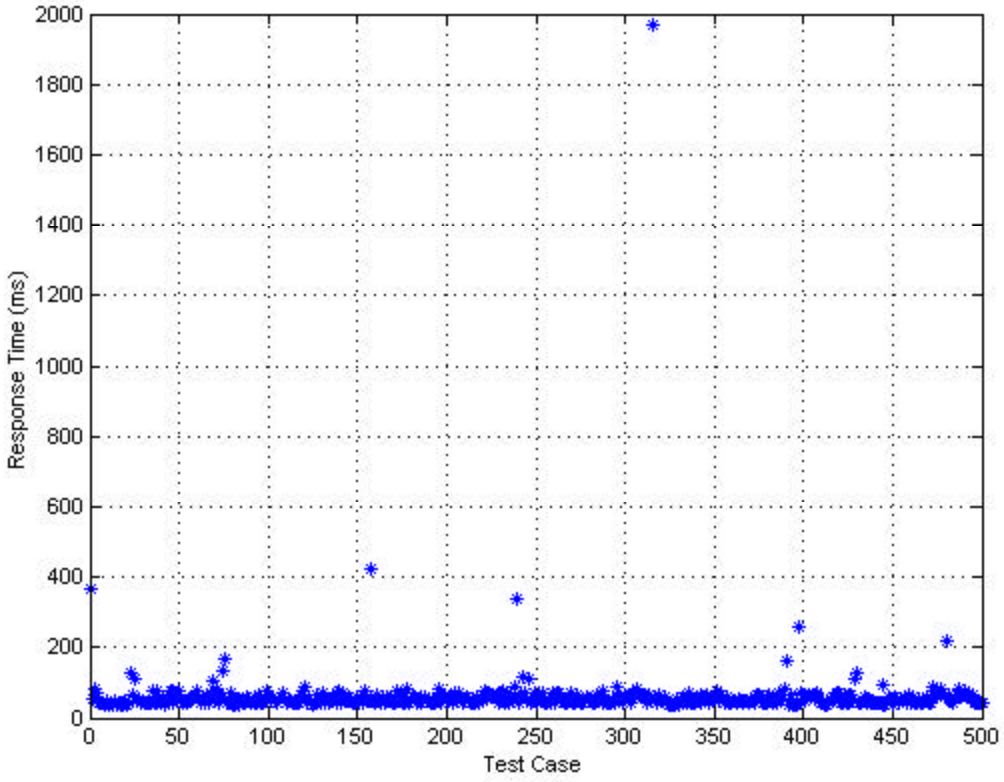

Figure 6.3b Response Time Diagram of second Client in Experiment 2

The first client response times changed between 50 – 600 ms while the second client response times grouped around 50 ms. The response times of the first client tends to settle around 40 ms but it had obviously more unpredictable response times than the second client. The maximum response time of the experiment was 1995 ms and occurred with the second client. This was interesting because the second client had relatively uniform response times compared to the first client. We might conclude that this was a result of the underlying network traffic and due to congestion occurring in the network during that particular attempt. But, as explained in experiment 1, the Turtle1 and Sun58 workstations were isolated from the rest of the network. This prevents us from concluding that this was a result of the network traffic. We conjecture that this unexpected high response time was a result of an overhead introduced by the operating systems on the workstations or by JavaSpaces itself.

3. Experiment 3

This experiment was same as the experiment 1 except that we used the read service instead of the write service of JavaSpaces. We used only one client, so the load on JavaSpaces was low.

A client, created by the test program on the Sun58 workstation, contacted the JavaSpaces server, which was running on Turtle1. The client attempted to read an entry 500 times from JavaSpaces. There were no other applications running on Turtle1 and Sun58 (except the JavaSpaces server and other system tasks).

The chart in Figure 6.4 shows the response times of the JavaSpaces server for each read attempt.

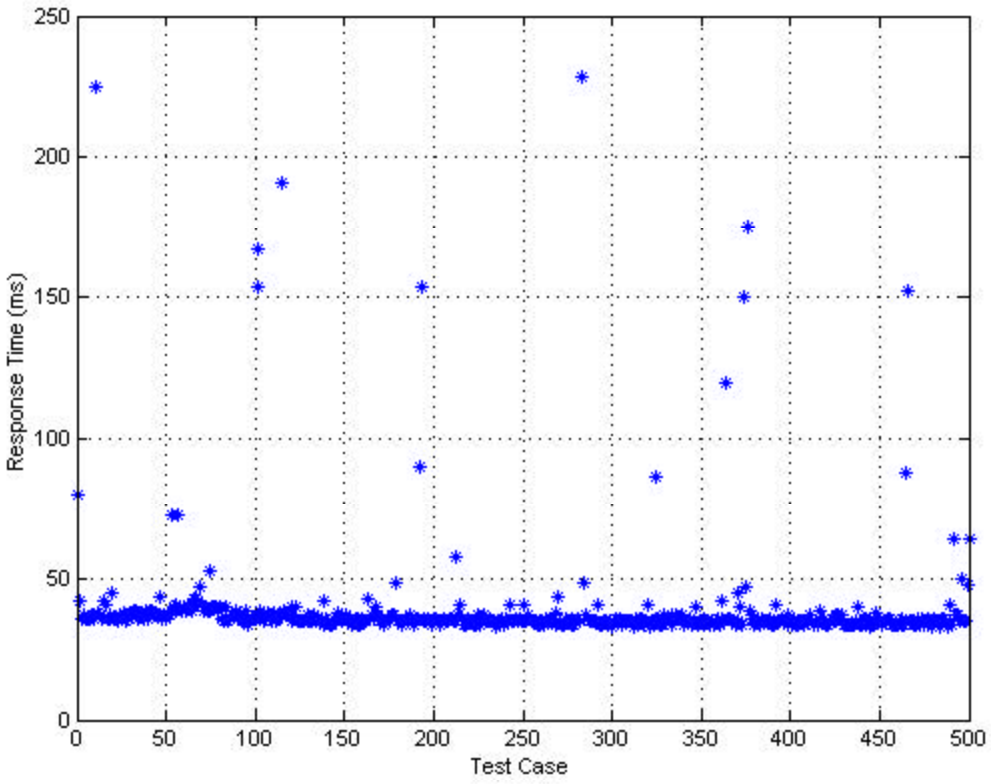

Figure 6.4 Response Time Diagram of Experiment 3

As expected, the response times of the JavaSpaces were similar to those of experiment 1. However, the response times were also more uniformly distributed than those of experiment 1 and we had relatively fewer response times that were higher than the average. We conjecture that this is the result of the overhead introduced by the operating systems or by JavaSpaces itself as in experiment 1.

4. Experiment 4

The goal and scenario of this experiment was the same as that of experiment 2 except that we used the read service instead of the write service of JavaSpaces. We created two clients on the Sun58 workstation to have the same load on JavaSpaces as that of experiment 2.

Two clients, created by the test program on the Sun58 workstation, contacted the JavaSpaces server, which was running on Turtle1. Both clients attempted to read an entry 500 times from the JavaSpaces. There were no other applications running on Turtle1 and Sun58 (except the JavaSpaces server and other system tasks).

The charts in Figure 6.5a and b show the response times of the JavaSpaces server for each client respectively. We had similar response time diagrams to those of experiment 2 for both clients.

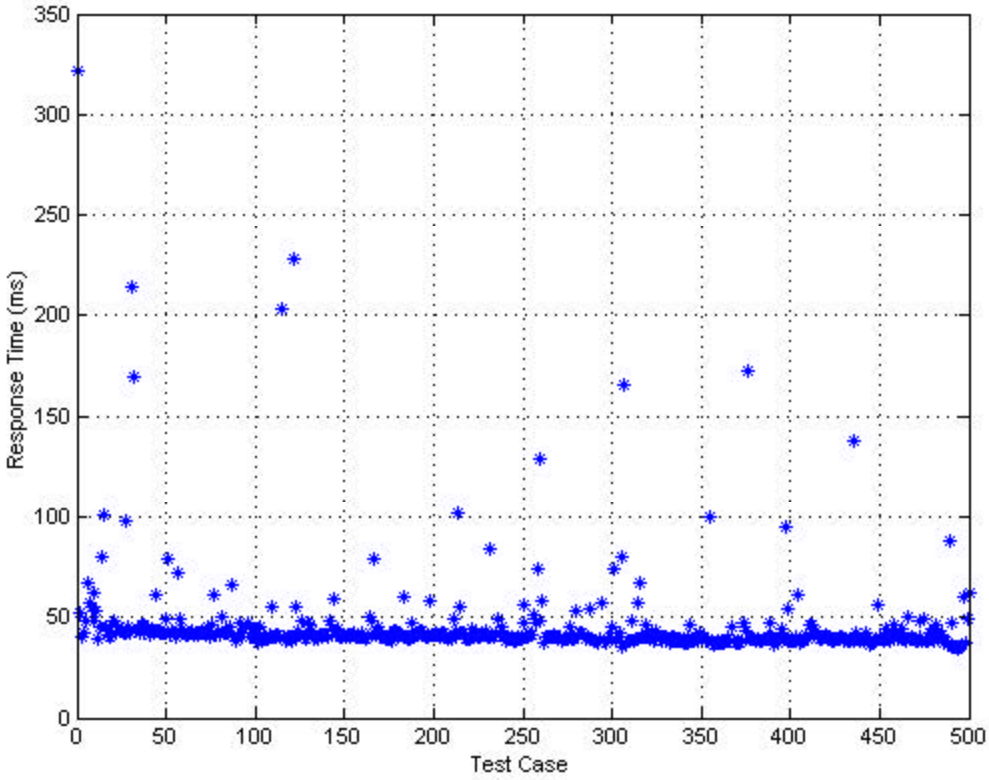

Figure 6.5a Response Time Diagram of first Client in Experiment 4

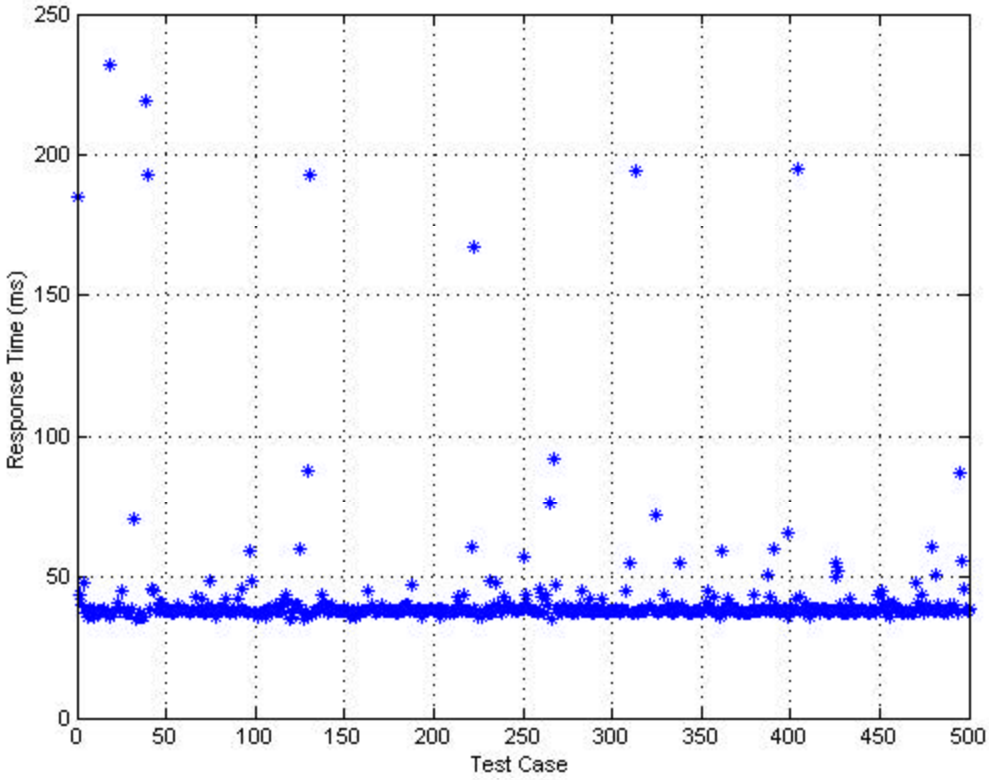

Figure 6.5b Response Time Diagram of second Client in Experiment 4

We conjectured that JavaSpaces has similar performance to those of write service for the read service, because we had similar response time diagrams for experiment 2 and 4 under the same load.

5. Experiment 5

This experiment was same as experiment 1 except that we used the Norma workstation. Our goal was to observe the performance of JavaSpaces with a different type of machine. We used only one client, so the load on JavaSpaces was low.

A client, created by the test program on the Norma workstation, contacted the JavaSpaces server, which was running on Turtle1. The client attempted to write an entry

500 times to JavaSpaces. There were no other applications running on the Turtle1 and Norma workstations (except the JavaSpaces server and other system tasks).

The chart in Figure 6.6 shows the response times of the JavaSpaces server for each write attempt.

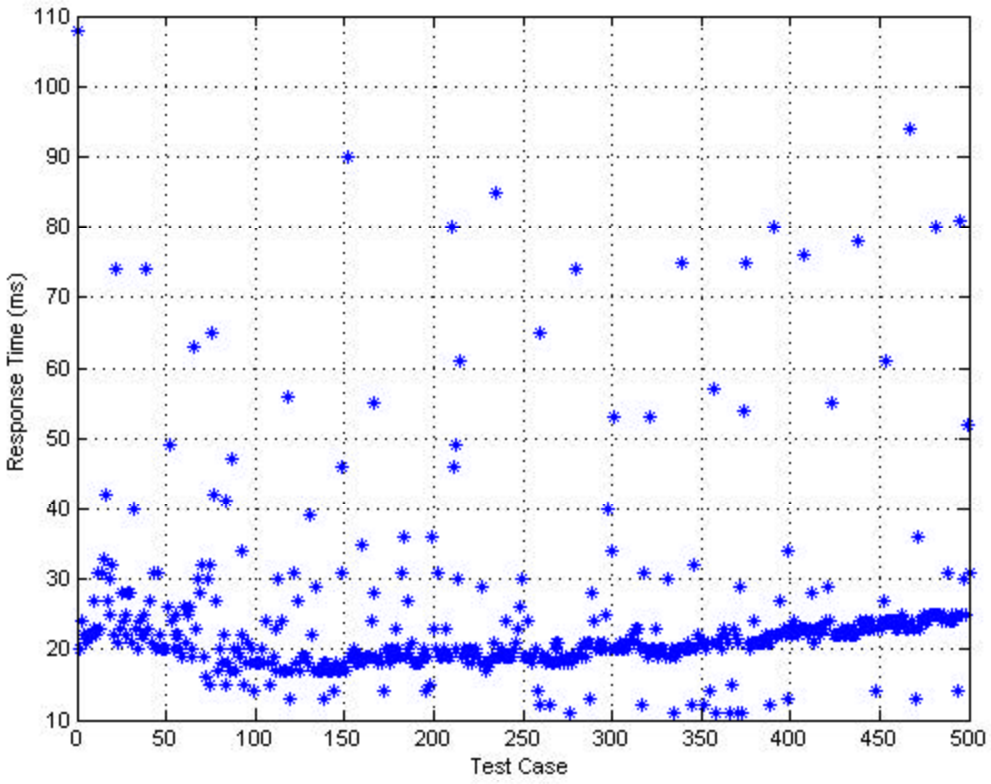

Figure 6.6 Response Time Diagram of Experiment 5

As seen in this chart, the response times of the JavaSpaces service grouped around 20 ms that is less than that of the Sun58 workstation, which resided on the same LAN with the JavaSpaces server. We also had lower response times than other experiments conducted with Sun58 workstation. We conjectured that the reason for this is that Norma workstation has a faster CPU (270 Mhz) and network connection (100 Mbps) than the Sun58 workstation (40 Mhz and 10 Mbps) and was thus able to overcome any communications latency presented by the fact that Norma was not isolated from the rest of the LAN.

We also observed that we had a lot of scattering in response times for this experiment. We conjectured that this was the result of a relatively high network traffic occurring during the experiment than those of other experiments.

6. Experiment 6

This experiment was similar to experiment 5 except that we measured the read service performance of JavaSpaces instead of write service. The goal of this experiment was to test the performance of the read service of JavaSpaces. We used only one client, so the load on JavaSpaces was low.

A client, created by the test program on the Norma workstation, contacted the JavaSpaces server, which was running on Turtle1. The client attempted to read an entry 500 times from JavaSpaces. There were no other applications running on the Turtle1 and Norma workstations (except the JavaSpaces server and other system tasks).

The chart in Figure 6.7 shows the response times of the JavaSpaces server for each read attempt.

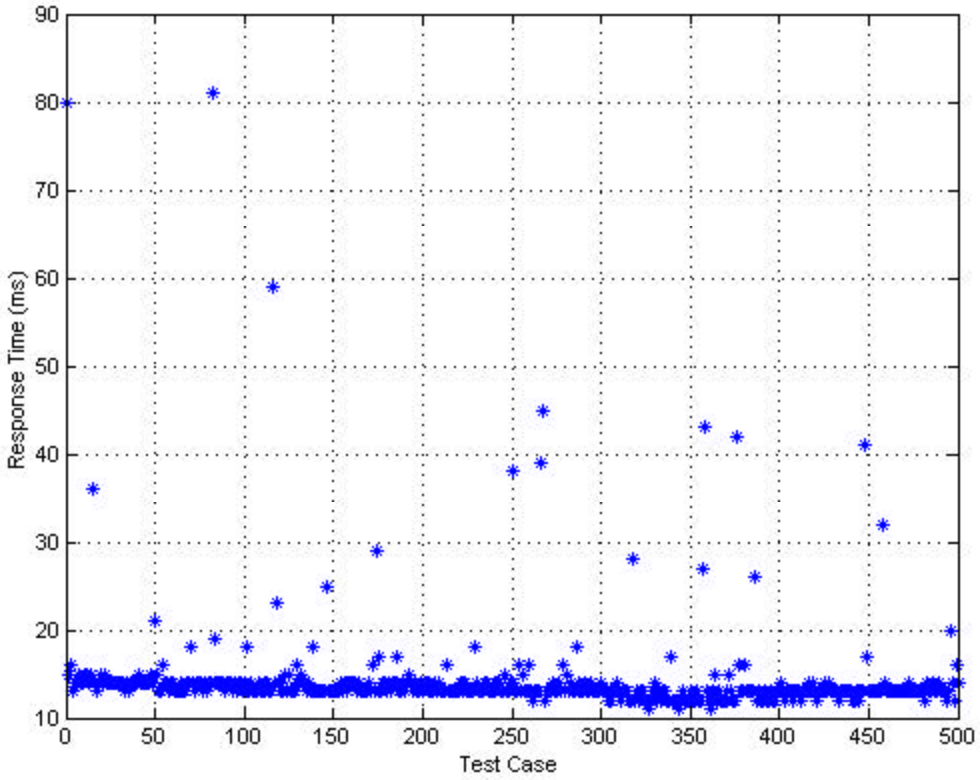

Figure 6.7 Response Time Diagram of Experiment 6

As in experiment 5, we observed that the response times of the JavaSpaces was relatively lower than that of experiments conducted with Sun58 workstation. We had also more uniformly distributed data than the experiment 5. We conjectured that the reason for relatively lower response times was the better CPU and network connection capabilities of Norma workstation than the Sun58 workstation as mentioned in experiment 5. We conjectured that the reason for more uniformly distributed data was relatively lower traffic in the network during this experiment than experiment 5.

7. Experiment 7

In this experiment, our goal was to measure the performance of JavaSpaces under relatively higher load. We used two different clients, each on a different machine (resided

on a separate LAN with JavaSpaces server), and attempted to use the JavaSpaces write service at the same time.

Two clients, created by the test programs, one on the Norma workstation and one on the Saturn workstation, contacted the JavaSpaces server, which was running on Turtle1. Both clients tried to write an entry 500 times to JavaSpaces at the same time. There were no other applications running on the Turtle1, Norma, and Saturn workstations (except the JavaSpaces server and other system tasks).

The charts in Figures 6.8a and b show the response times of the JavaSpaces server for each write attempt for the clients on the Norma and Saturn workstations.

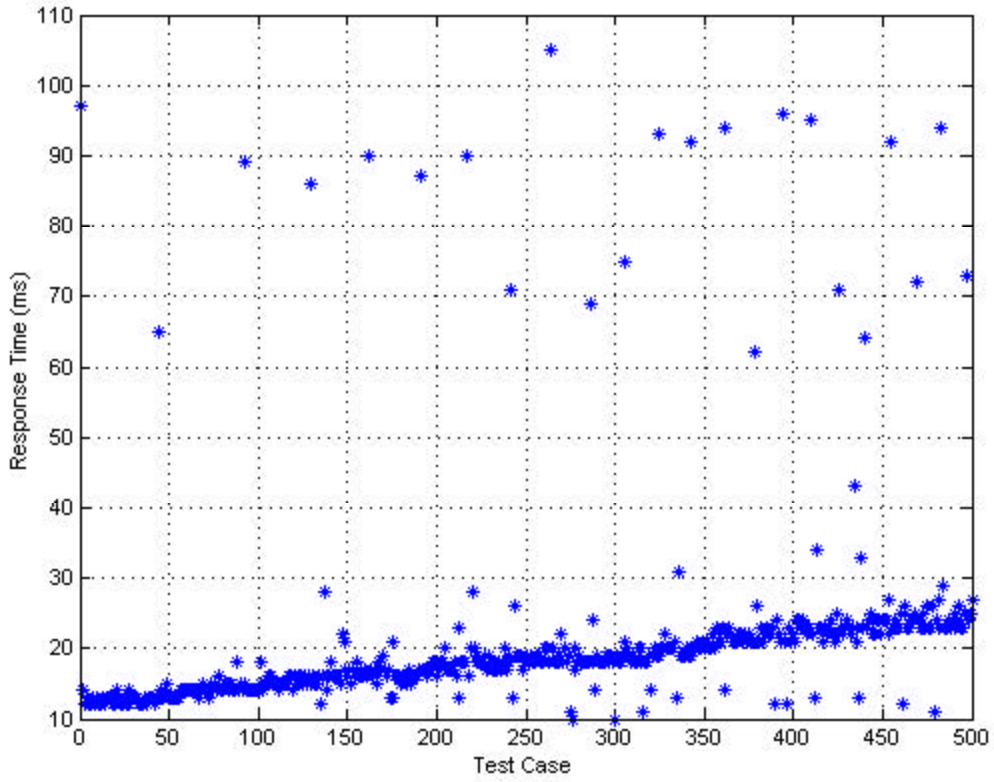

Figure 6.8a Response Time Diagram of Client on the Norma in Experiment 7

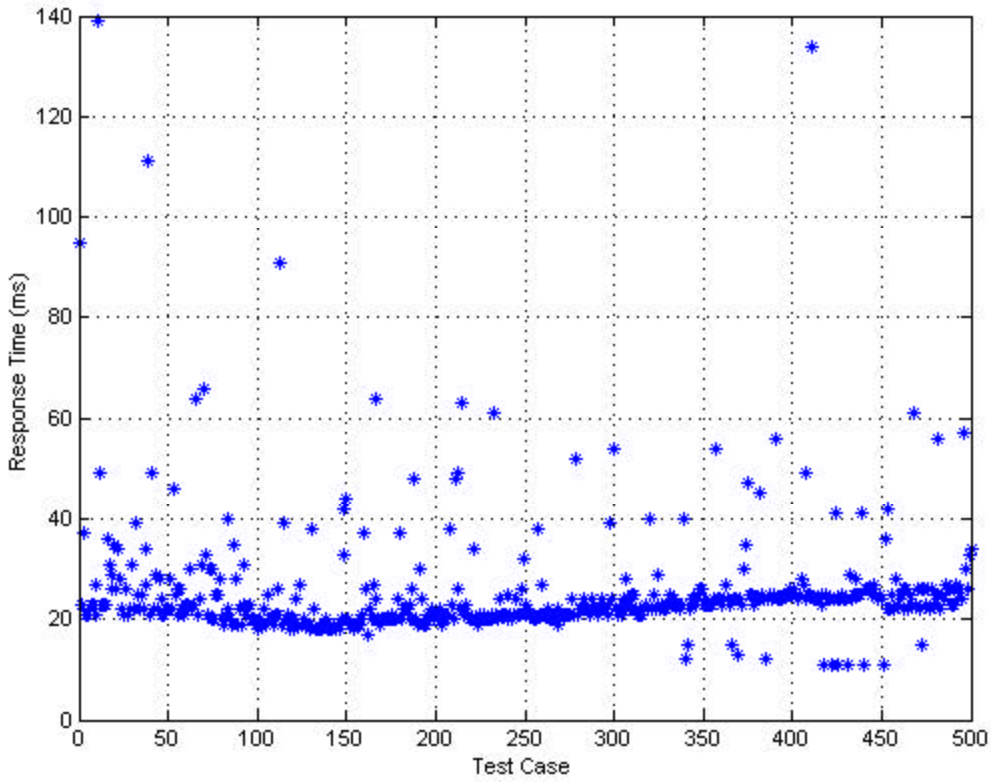

Figure 6.8b Response Time Diagram of Client on the Saturn in Experiment 7

As seen in Figure 6.8a, the response time diagram of the Norma workstation had an interesting upward trend. It also had unexpectedly higher response times than the Saturn workstation. The Saturn workstation also had an upward trend between attempts 150 – 440. This experiment was the first experiment we ran two different clients concurrently on two different machines. If noticed, we also had two clients running concurrently in experiment 4. But those clients were running on the same machine, so they were actually not running concurrently: they were simply using the same CPU. We conjectured that this was the reason of the upward trend in Norma workstation response times. The Saturn workstation response times were more uniformly distributed than the Norma workstation. We conjectured that JavaSpaces service was trying to service both clients equally and as a result of this, there was an upward trend in the response times of the Norma workstation going up to around 25 ms. Note that the Saturn workstation

response times starts from 25 – 30 ms, and then goes down to 18 ms while the Norma workstation response times starts from 14 ms and then goes up to 20 ms. At the end of diagrams, we can see that each client response times are around 25 ms and they are tend to be remain the same. As mentioned, both clients ran concurrently and so we cannot predict the order they contact to the JavaSpaces server. We conjectured that when we started the experiment, the client on the Norma workstation contacted the JavaSpaces server before the client on the Saturn workstation. As a result of this, the first attempts of the Norma workstation had relatively lower response times than those of the Saturn workstation. When JavaSpaces server noticed that two different clients were trying to use write service, it started to equally service to both clients. As a result of this, response times of the Norma workstation tended to go upward while the Saturn workstation response times tended to go downward.

We may also argue that the reason for the upward trend may be the JavaSpaces write service. When a client wants to write a new Entry to JavaSpaces, JavaSpaces must allocate new memory space for the Entry object. JavaSpaces places each write attempt in a queue if it cannot process them at the moment. Concurrent or high load on JavaSpaces may cause this unexpected upward trend as a result of this queuing.

8. Experiment 8

The purpose of this experiment was to test the performance of JavaSpaces service under relatively higher loads. We tested the write service of JavaSpaces in this experiment. We used three clients, each running on a different machine to increase the load.

Three clients, created by the test programs, one on the Norma workstation, one on the Saturn workstation, and one on the Moon workstation, contacted the JavaSpaces server, which was running on Turtle1. All clients attempted to write an entry 500 times to the JavaSpaces at the same time. There were no other applications running on Turtle1 and other machines used in this experiment (except the JavaSpaces server and other system tasks).

The charts in Figures 6.9a, b, and c show the response times of the JavaSpaces server for each write attempt for the clients on the Norma, Saturn, and Moon workstations.

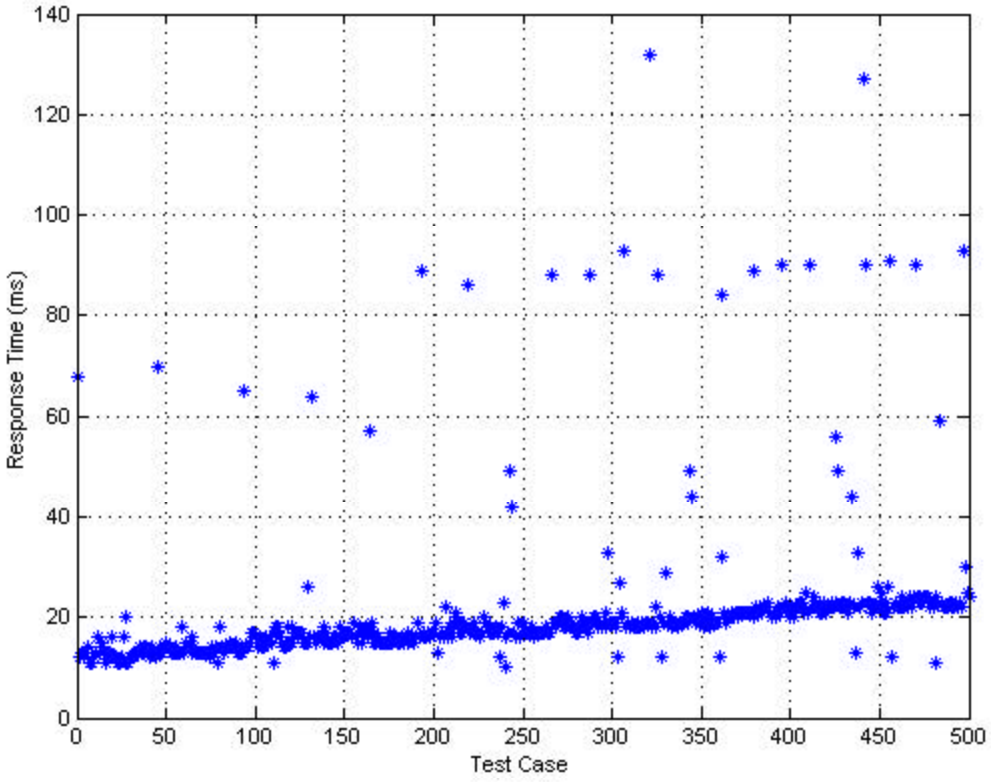

Figure 6.9a Response Time Diagram of Client on the Norma in Experiment 8

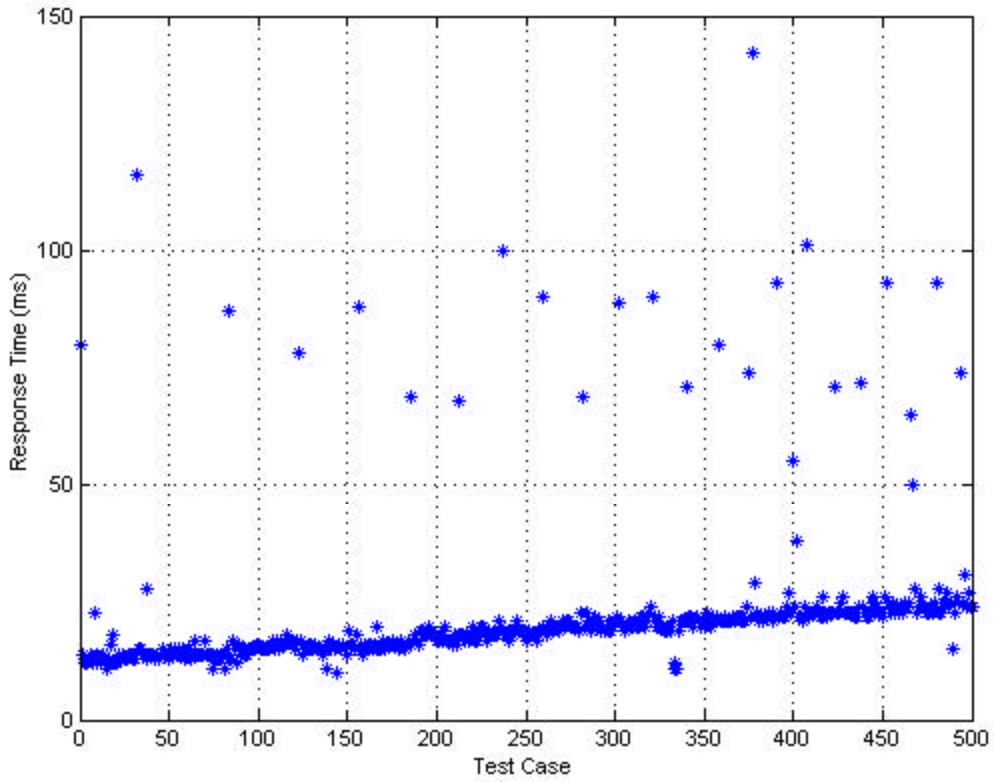

Figure 6.9b Response Time Diagram of Client on the Saturn in Experiment 8

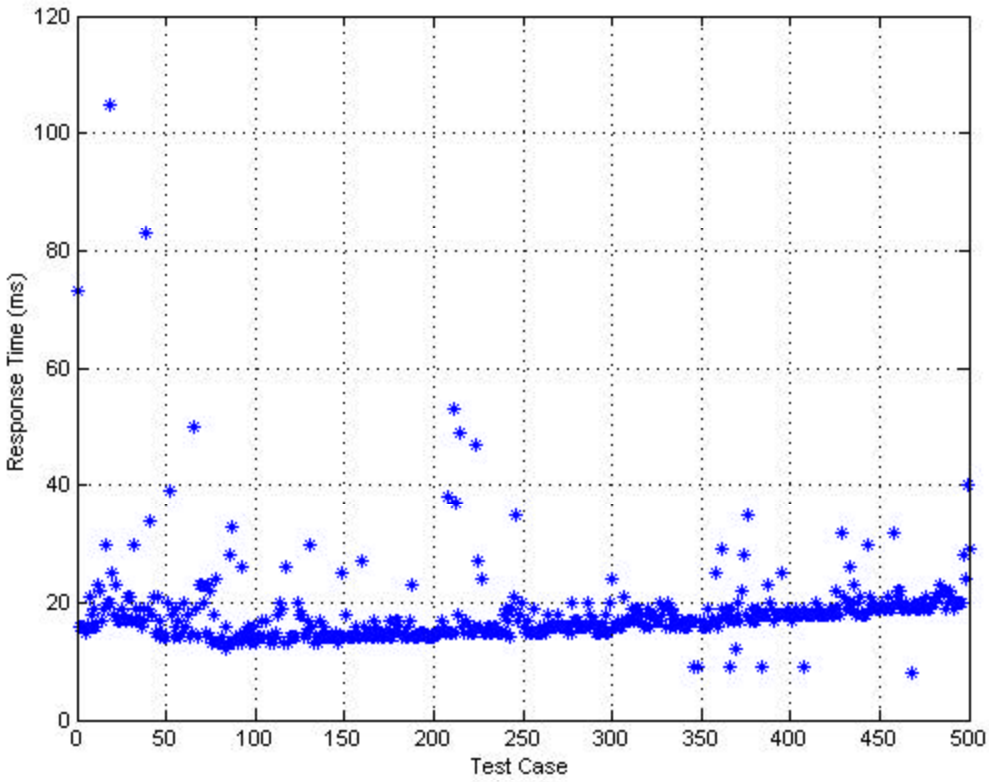

Figure 6.9c Response Time Diagram of Client on the Moon in Experiment 8

The response times of all clients were similar to each other, and uniformly distributed. This means that the JavaSpaces service responded to each client equally. It is also important to observe that under relatively high load (three concurrent clients), the average response time of the JavaSpaces service is very low (around 20 ms). We conjectured that JavaSpaces write service was not affected from the increasing load. Note that the response time diagram of all clients had an upward trend. We conjectured that the reason was the same as that of experiment 7.

9. Experiment 9

This experiment was the same as the experiment 8 except that we tested the read service instead of the write service of JavaSpaces. We used three clients, so the load on JavaSpaces was high.

Three clients created by the test programs, one on the Norma workstation, one on the Saturn workstation, and one on the Moon contacted the JavaSpaces server, which was running on Turtle1. All clients attempted to read an entry 500 times from JavaSpaces at the same time. There were no other applications running on Turtle1 and the other machines (except the JavaSpaces server and other system tasks).

The charts in Figures 6.10a, b, and c show the response times of the JavaSpaces server for each read attempt for the clients on the Norma, Saturn, and Moon workstations.

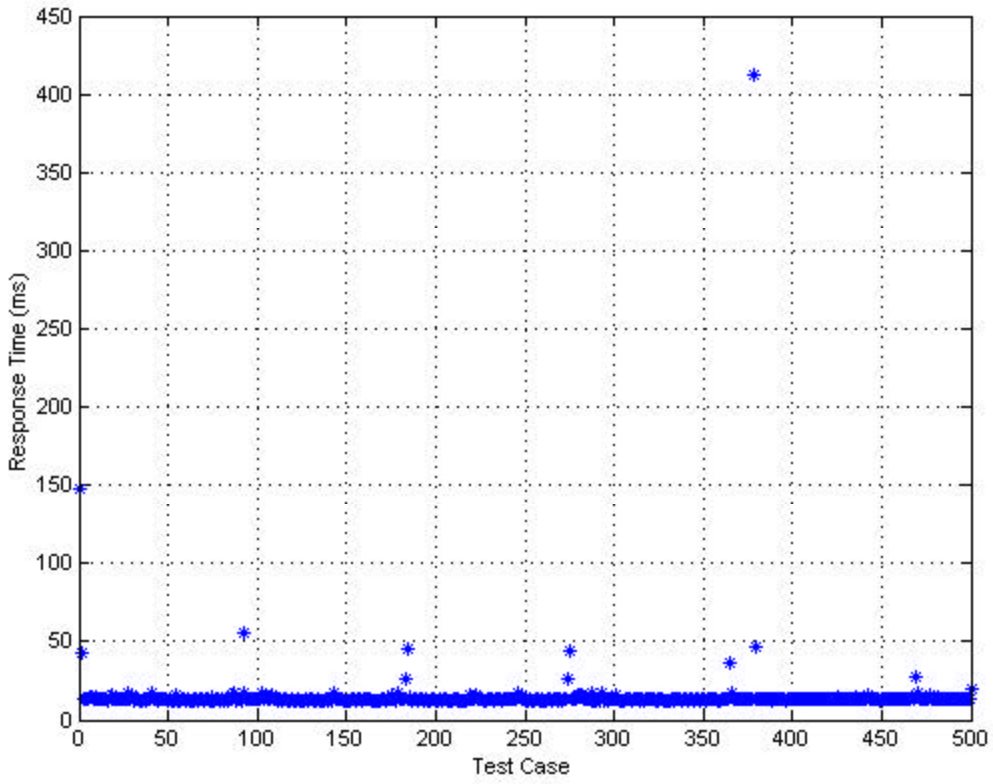

Figure 6.10a Response Time Diagram of Client on the Norma in Experiment 9

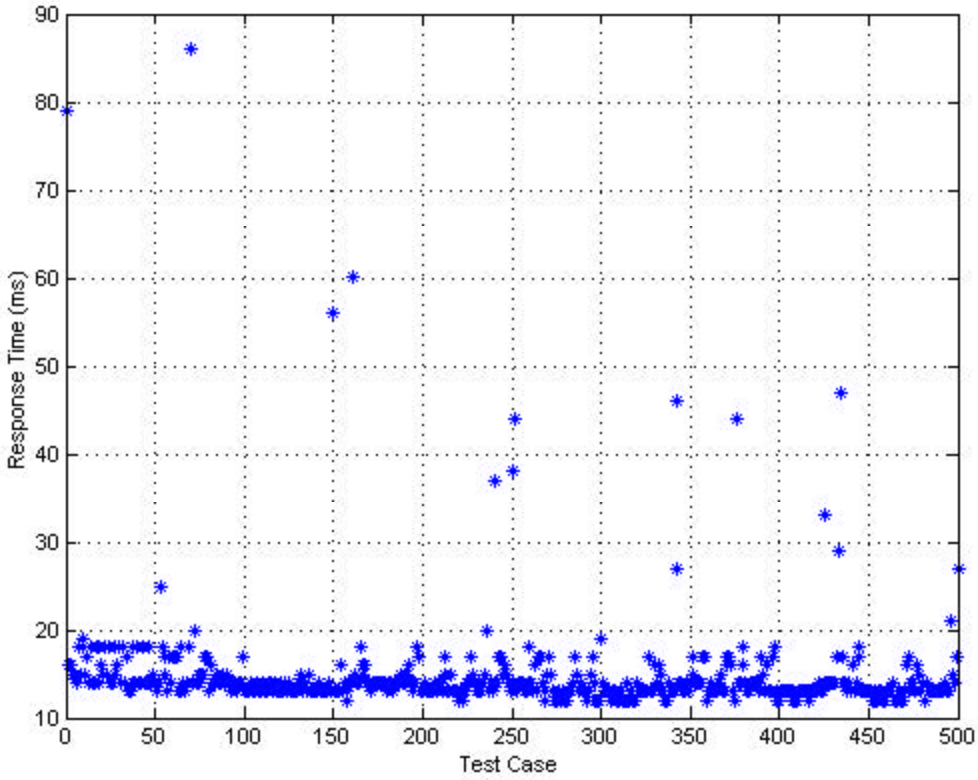

Figure 6.10b Response Time Diagram of Client on the Saturn in Experiment 9

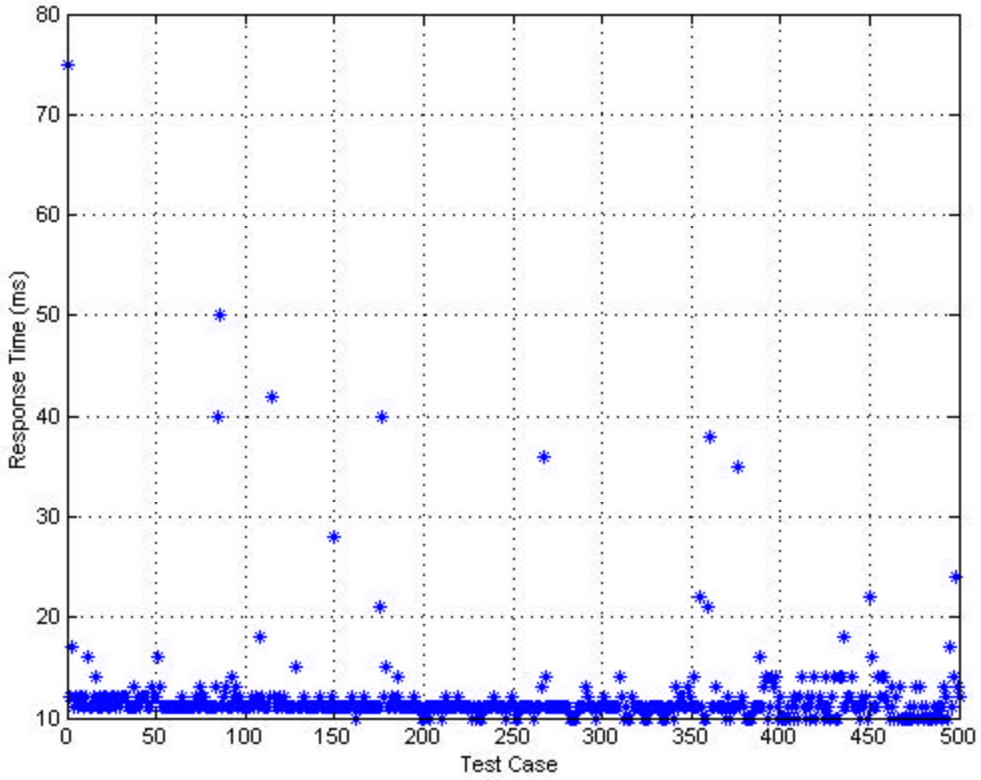

Figure 6.10c Response Time Diagram of Client on the Moon in Experiment 9

The result of this experiment was similar to the result of the experiment 8. We observed that all clients had a similar response times and that the client on the Moon workstation had relatively shorter response times. Another important observation was that the overall response times for the read service in this experiment was shorter than the write service in experiment 8.

We did not observe the upward trend in experiment 7 and 8 in this experiment even though that this experiment's scenario was same as those of experiment 7 and 8 (different client running concurrently on different machines). We conjecture that the reason of this was the read service. JavaSpaces does not allocate new memory space to process the read services. It only creates a new copy of an existing Entry object in the space (if there is an matching Entry object). We had concluded two different reasons for this unexpected upward trend in experiment 7. The observation we made for this

experiment strengthens our second conclusion. The reason for this upward trend may be the implementation of the JavaSpaces write and read services. As mentioned, JavaSpaces must allocate new memory for each write service but it does not need to allocate new memory space to implement read service and so it does not introduce an overhead.

10. Experiment 10

The goal of this experiment was to test the performance of JavaSpaces write service for a local client. A second, non-local client was used to increase the load on JavaSpaces.

Two clients created by the test programs, one on Turtle1 and one on the Sun58 workstation, contacted the JavaSpaces server, which was running on Turtle1. The clients attempted to write an entry 500 times to JavaSpaces at the same time. There were no other applications running on the Turtle1 and Sun58 workstations (except the JavaSpaces server and other system tasks).

The charts in Figures 6.11a, and b show the response times of the JavaSpaces server for each write attempt for both clients.

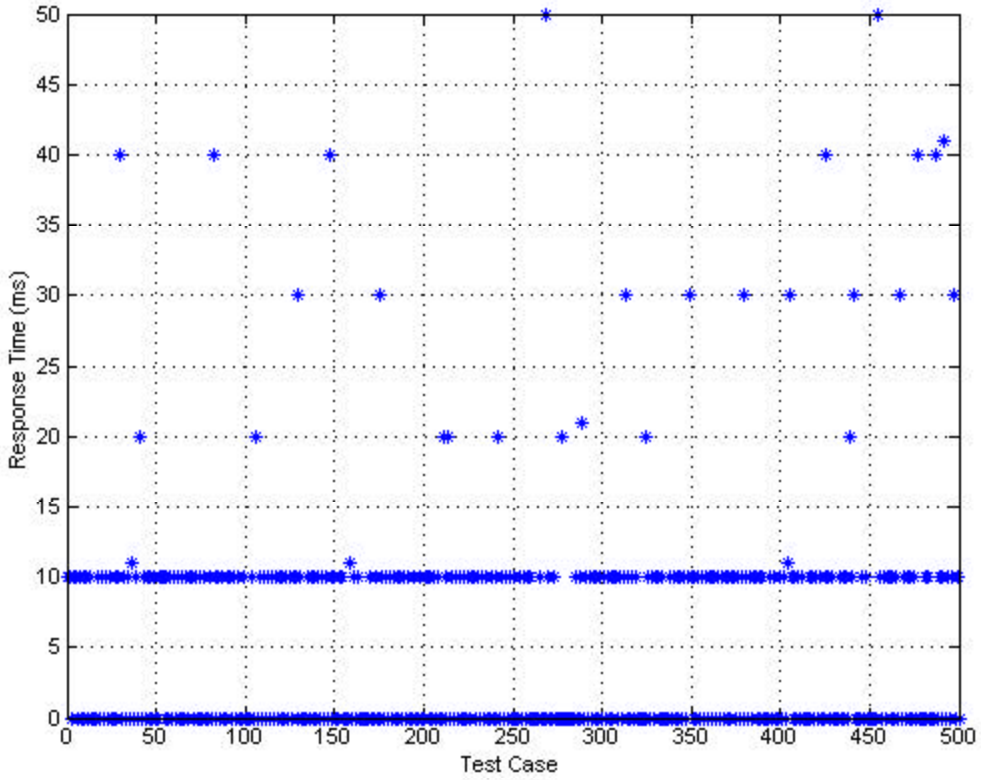

Figure 6.11a Response Time Diagram of Client on the Turtle1 in Experiment 10

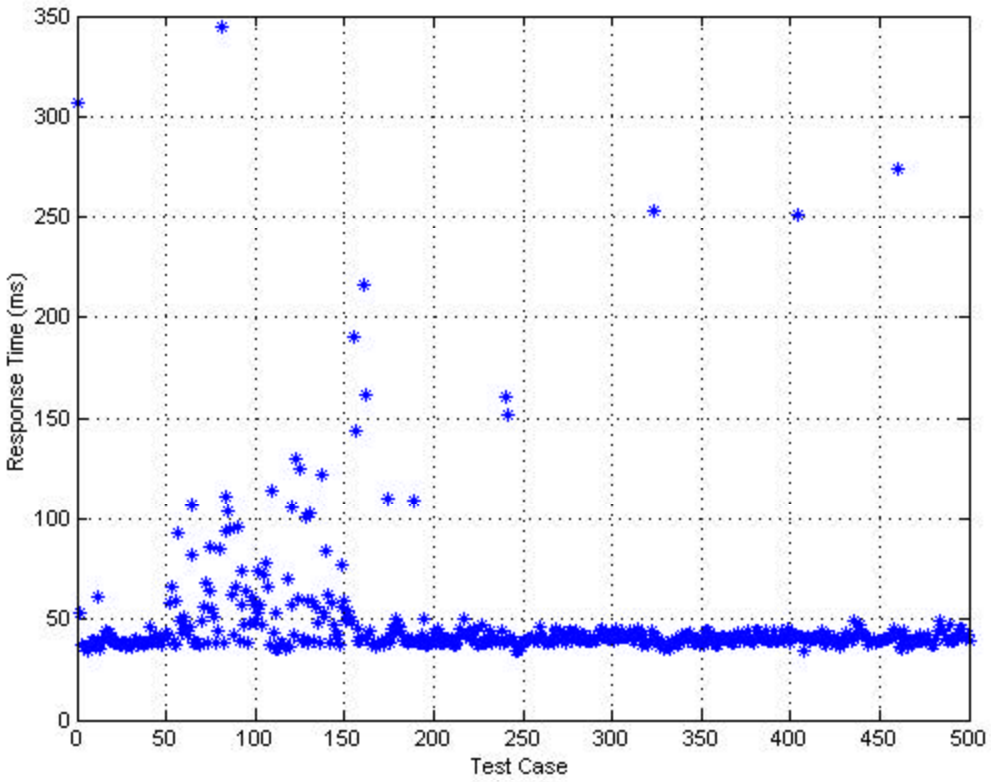

Figure 6.11b Response Time Diagram of Client on the Sun58 in Experiment 10

We observed that for the local client, response times of the JavaSpaces server were often ~0 ms. We observed a periodic change in response times between ~0 and 10 ms. The reason for this was that both the local client and the JavaSpaces service competed for the same CPU. They used the CPU periodically because they each had the same priority. There were no other applications running on Turtle1 during the experiment. Also, we observed some surprisingly longer response times (~50 ms) for the non-local client. For the non-local client, the response times were similar to that of previous experiments conducted using Sun58 workstation. There was an unexpected scattering between attempts 50 and 150 in the response times for the non-local client. Because the Sun58 workstation and Turtle1 were isolated from the rest of the network, we conjecture that the reason for this scattering might be the delay introduced by the operating system of the Sun58 workstation.

11. Experiment 11

This experiment was conducted to measure the average response time of the JavaSpaces read service for a bcal client. We also added a second, non-local client on Sun58 workstation, to increase the load on JavaSpaces.

Two clients created by the test programs, one on Turtle1 and one on the Sun58 workstation, contacted the JavaSpaces server, which was running on Turtle1. The clients attempted to read an entry 500 times from JavaSpaces at the same time. There were no other applications running on the Turtle1 and Sun58 workstations (except the JavaSpaces server and other system tasks).

The charts in Figures 6.12a, and b show the response times of the JavaSpaces server for each read attempt for the clients on the Turtle1 and Sun58 workstations.

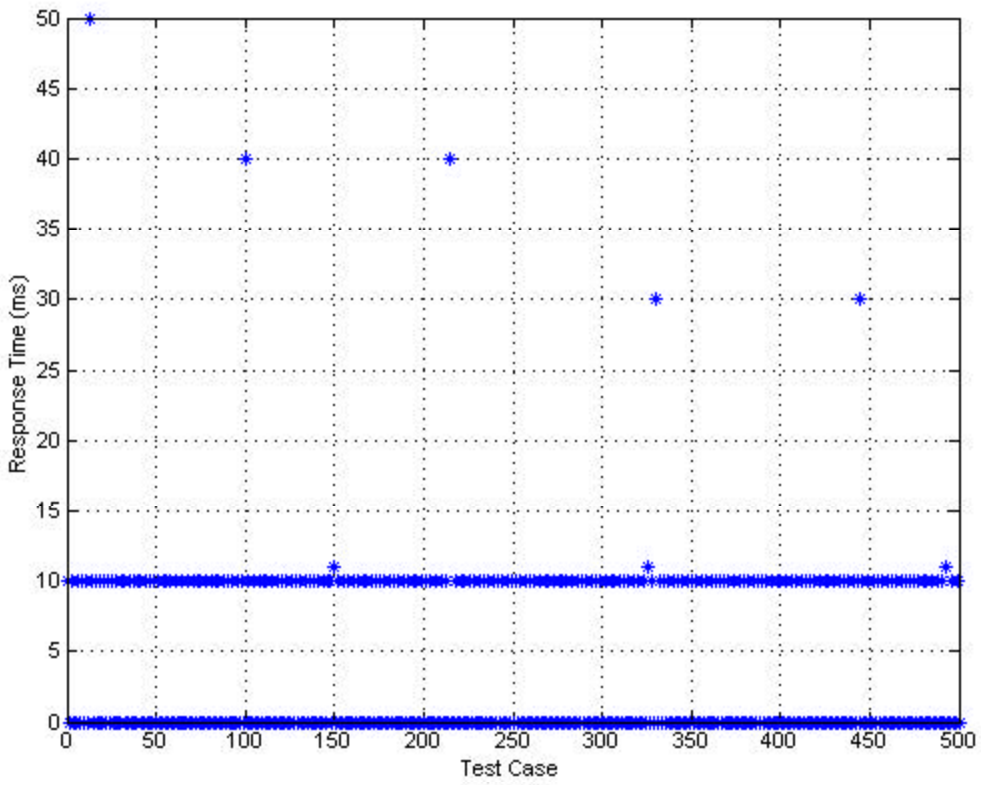

Figure 6.12a Response Time Diagram of Client on the Turtle1 in Experiment 11

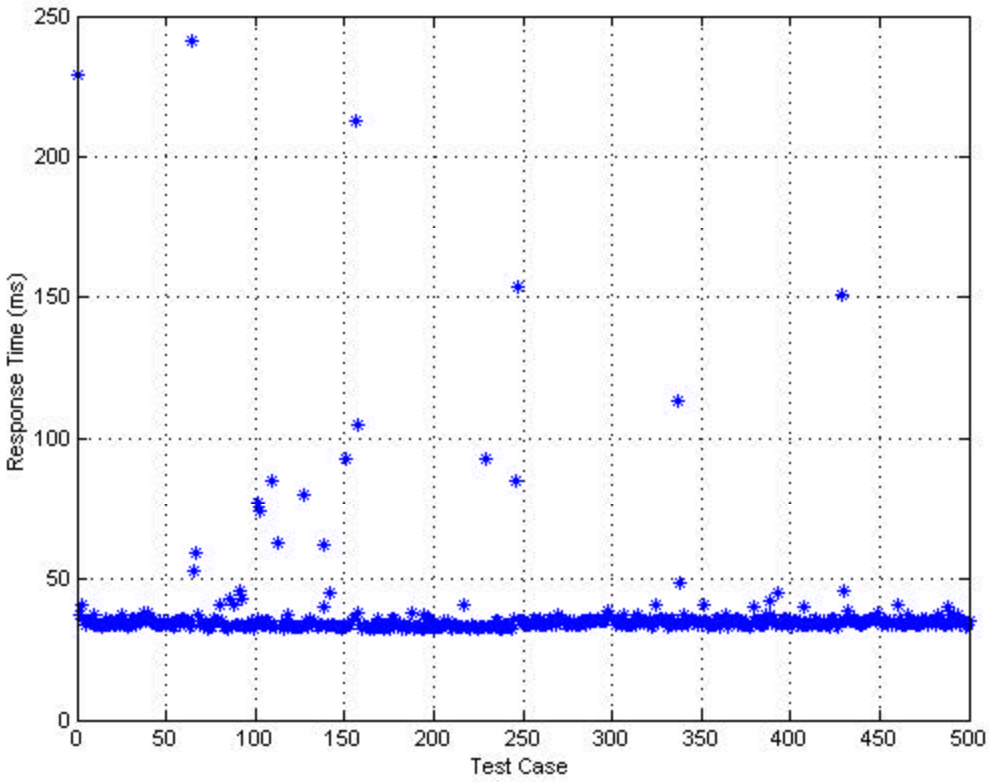

Figure 6.12b Response Time Diagram of Client on the Sun58 in Experiment 11

The results of this experiment were similar to the results of experiment 10. We observed a periodical switch between ~0 and 10 ms in response times of the local client. The second, non-local client, which ran on the Sun58 workstation, had similar response times as those in previous experiments conducted with the Sun58 workstation.

12. Experiment 12

The purpose of this experiment was to test the performance of JavaSpaces write service under relatively higher loads. We also aimed to observe the relative response times of the clients running concurrently on the same machines. We used four clients and two machines for this experiment. We ran two clients concurrently on each machine to increase the load on JavaSpaces.

Four clients created by the test programs, two on Turtle1 and two on the Sun58 workstation, contacted the JavaSpaces server, which was running on Turtle1. Each client attempted to write an entry 500 times to JavaSpaces at the same time. There were no other applications running on the Turtle1 and Sun58 workstations (except the JavaSpaces server and other system tasks).

The charts in Figures 6.13a, b, c, and d show the response times of the JavaSpaces server for each read attempt for the clients on the Turtle1 and Sun58 workstations.

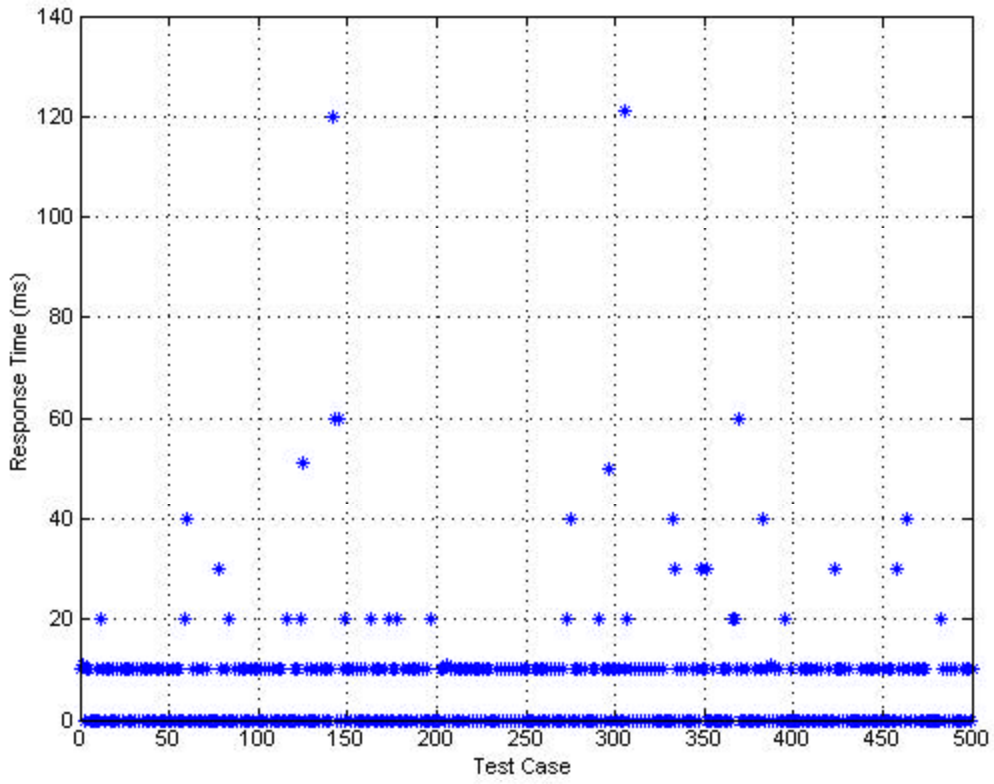

Figure 6.13a Response Time Diagram of first Client on the Turtle1 in Experiment 12

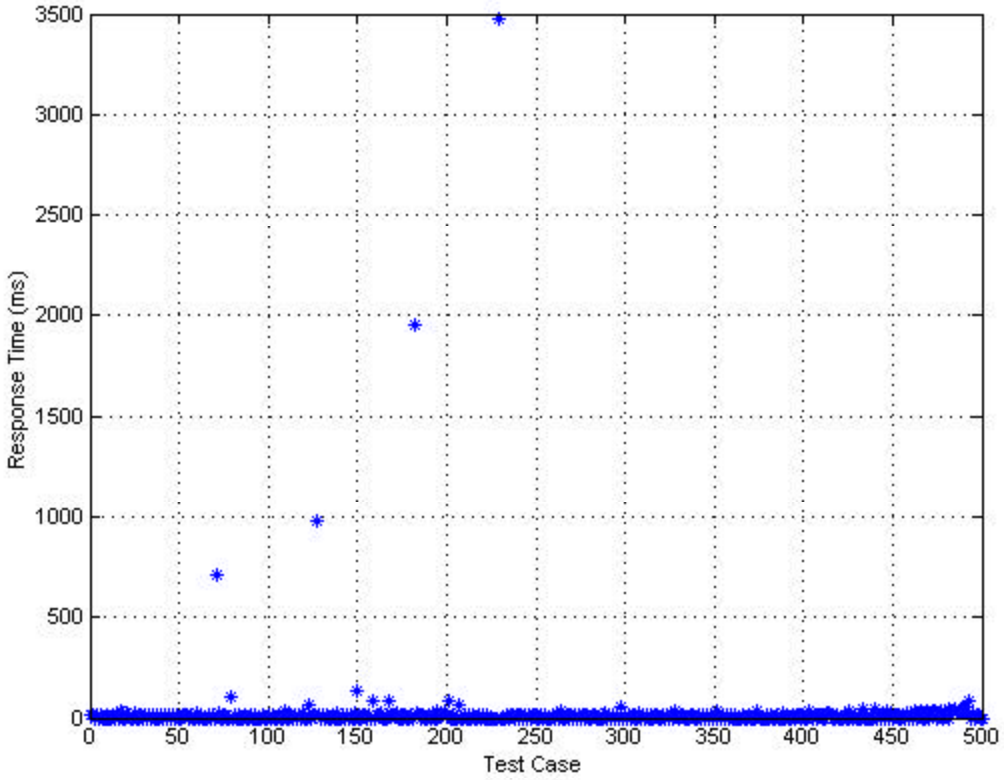

Figure 6.13b Response Time Diagram of second Client on the Turtle1 in Experiment 12

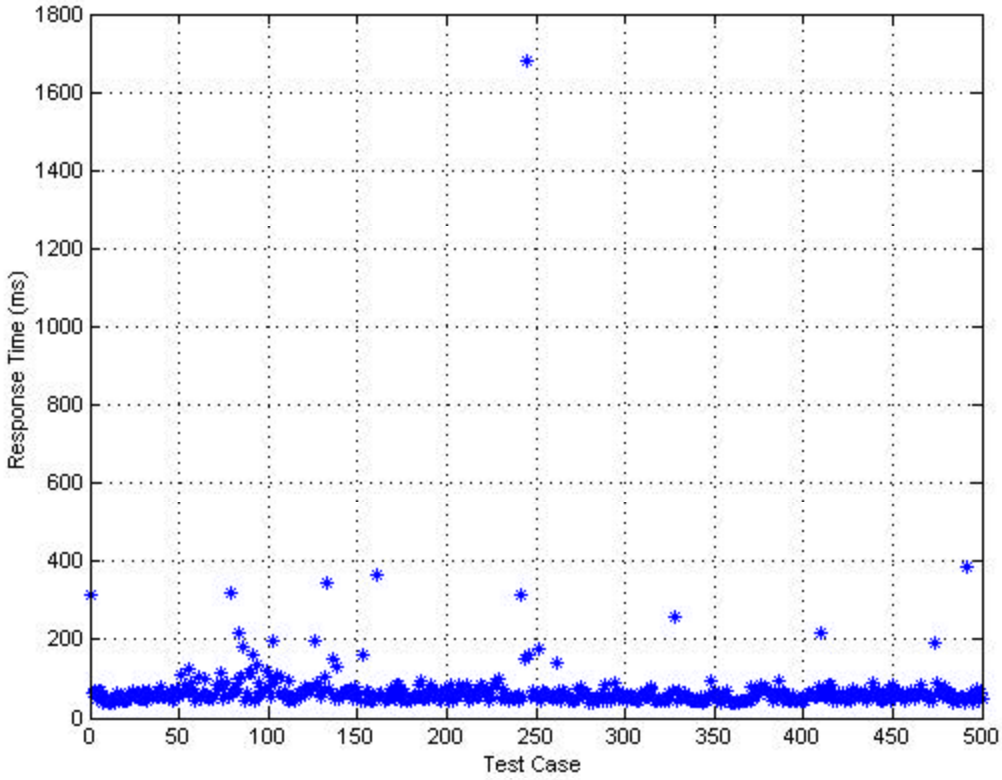

Figure 6.13c Response Time Diagram of first Client on the Sun58 in Experiment 12

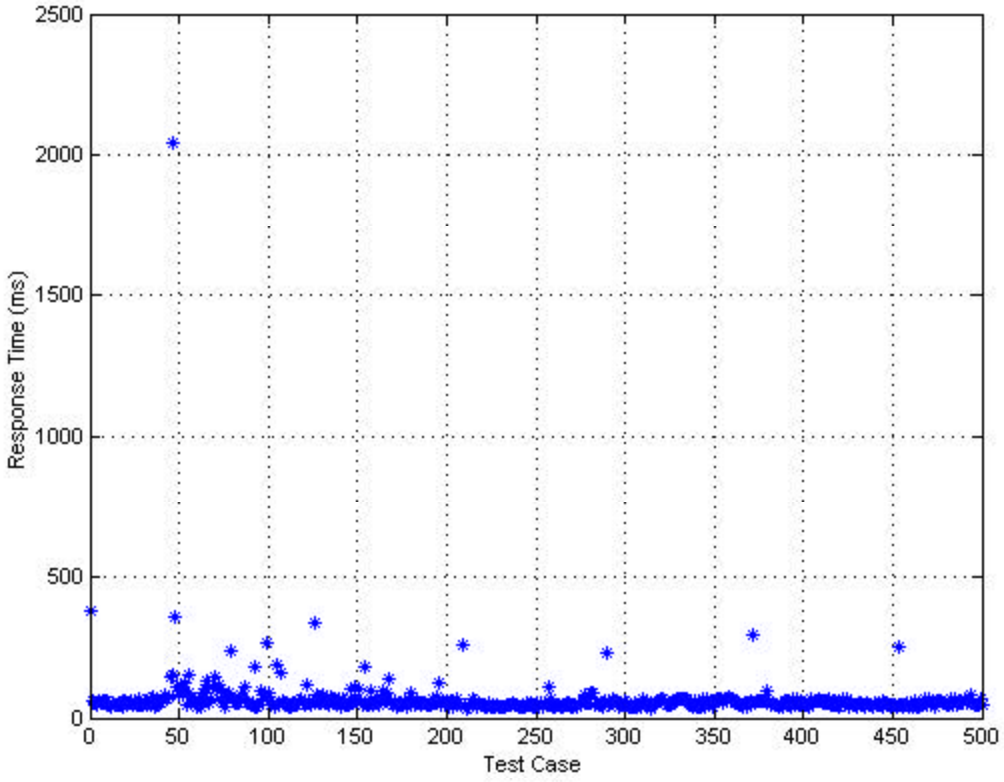

Figure 6.13d Response Time Diagram of second Client on the Sun58 in Experiment 12

The results were similar to the previous experiments conducted with Turtle1 and Sun58 workstation. An important observation was that there were some unexpectedly high response times for both machines (e.g. 120 ms on Turtle1 client 1, 3500 ms on Turtle1 client 2, 1600 ms on Sun58 client 1, and 2000 ms on Sun58 client2). We conjecture that the reason for this was the relatively higher load on the JavaSpaces service.

13. Experiment 13

This experiment was similar to experiment 12 except that we used the read service instead of write service of JavaSpaces. Our purpose was to test the performance of the JavaSpaces for the read service under relatively the highest load.

Four clients created by the test programs, two on Turtle1 and two on the Sun58 workstation, contacted the JavaSpaces server, which was running on Turtle1. Each client attempted to read an entry 500 times from JavaSpaces at the same time. There were no other applications running on the Turtle1 and Sun58 workstations (except the JavaSpaces server and other system tasks).

The charts in Figures 6.14a, b, c, and d show the response times of the JavaSpaces server for each read attempt for the clients on the Turtle1, and Sun58 workstations.

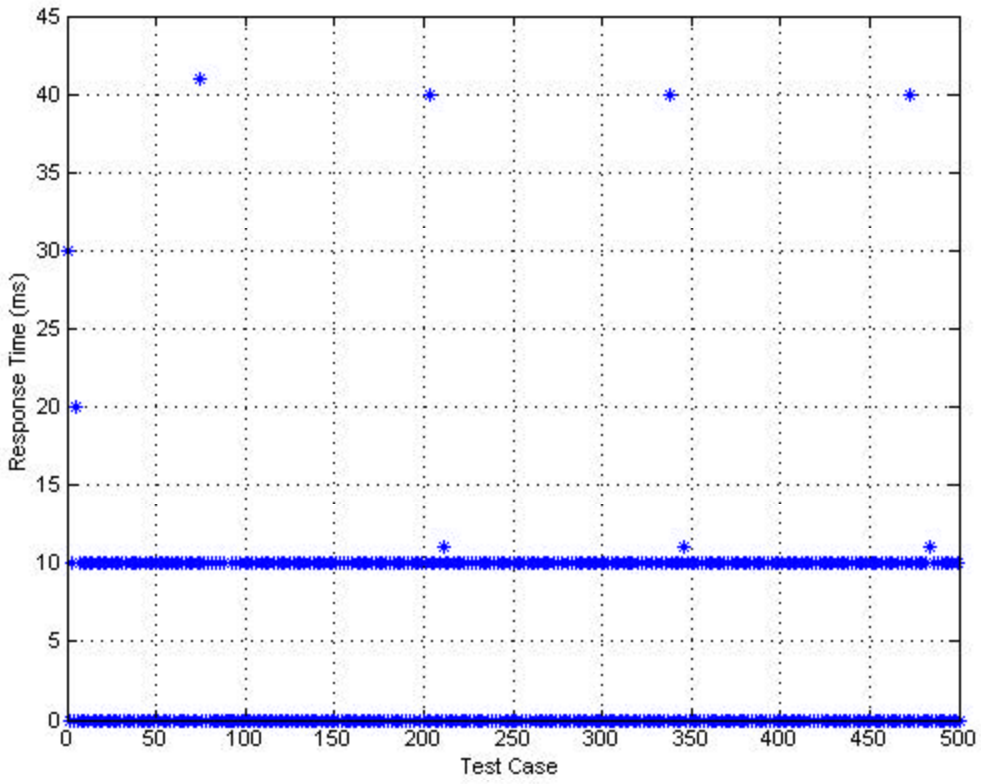

Figure 6.14a Response Time Diagram of first Client on the Turtle1 in Experiment 13

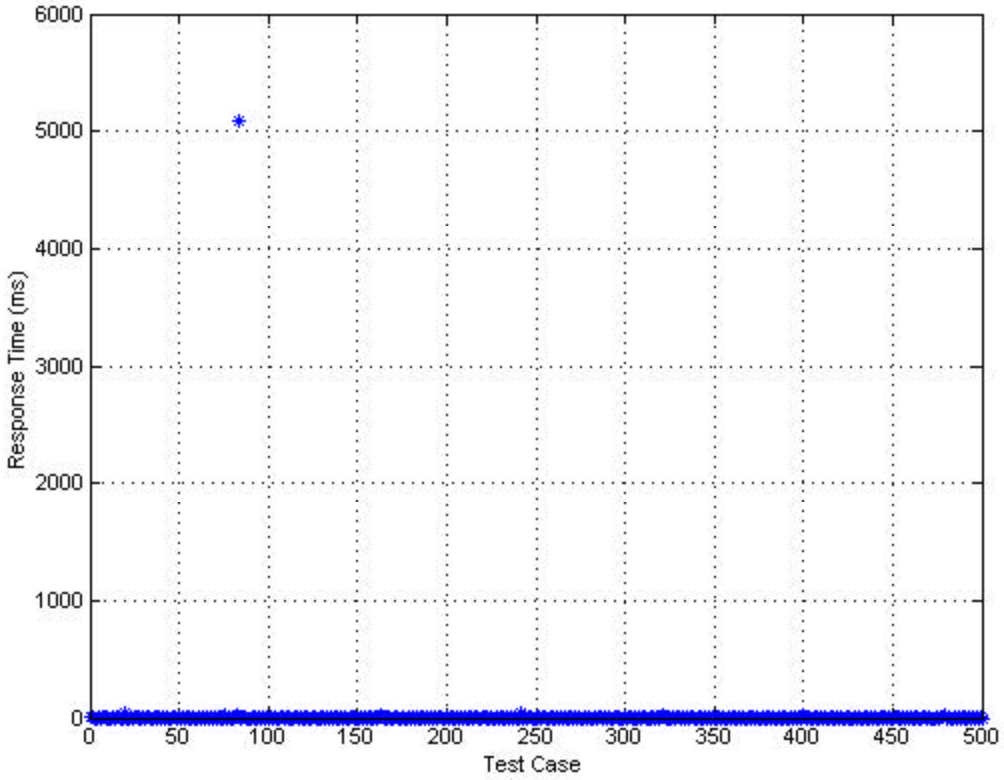

Figure 6.14b Response Time Diagram of second Client on the Turtle1 in Experiment 13

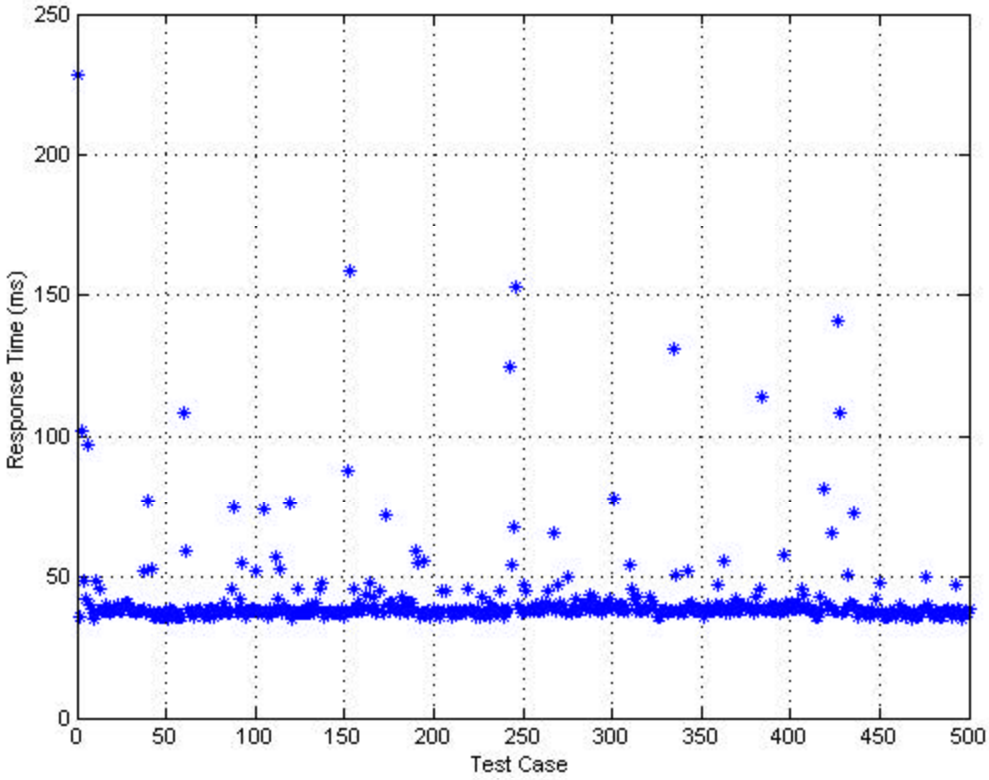

Figure 6.14c Response Time Diagram of first Client on the Sun58 in Experiment 13

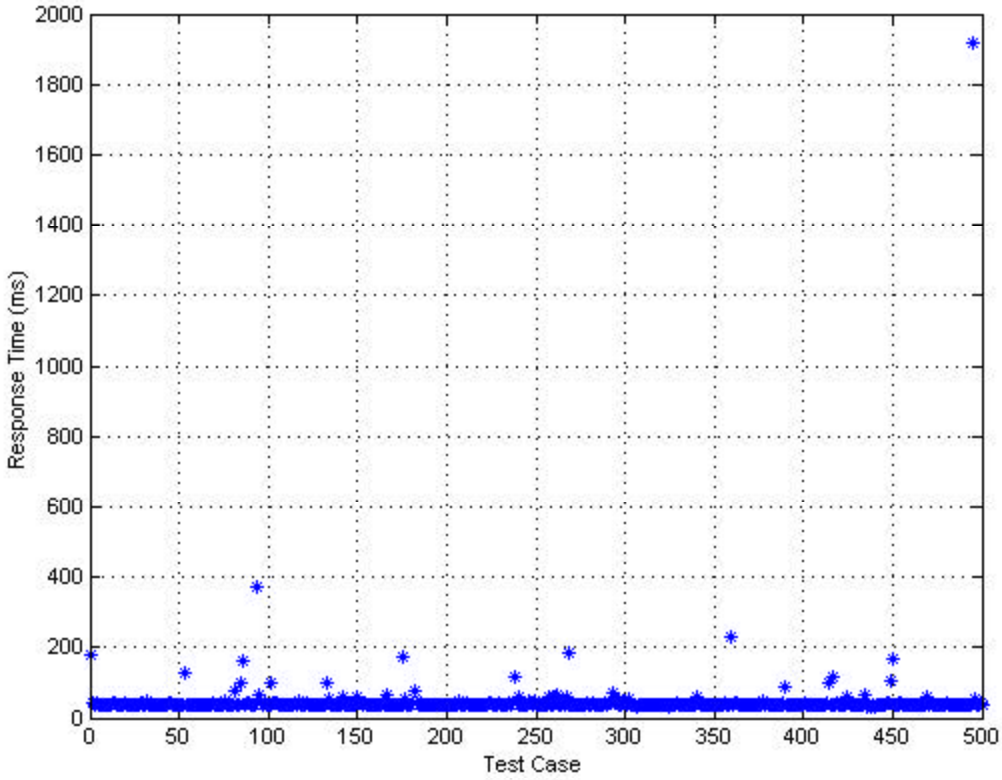

Figure 6.14d Response Time Diagram of second Client on the Sun58 in Experiment 13

The response times were similar to the experiment 12. We also observed that for each client, there was an unexpectedly high response time similar to that of experiment 12. We conjecture that this was the result of the relatively higher load on the JavaSpaces. We might conclude that these unexpectedly high response times were the result of congestion in the underlying network; but, the similar behavior of the clients for both experiment 12 and 13 and isolation of the Turtle1 and Sun58 workstations from the rest of the network confirms our conclusion about the overhead introduced by JavaSpaces.

C. RESULTS

The data statistics obtained in the experiments are summarized in Table 7.2. In this table, we summarized the minimum/maximum, mean, median, standard deviation

and range between the minimum and maximum response times obtained in the experiments.

Experiment	Machine	Min (ms)	Max (ms)	First Res. Time (ms)	Mean (ms)	Std.Dev. (ms)	Range (ms)	Service
1	Sun58	36	456	456	58.63	40.14	420	write
2	Sun58	36	592	592	76.08	56.44	556	write
2	Sun58	36	1970	366	61.12	91.22	1934	write
3	Sun58	33	228	80	39.58	20.32	195	read
4	Sun58	35	232	185	42.32	20.83	197	read
4	Sun58	34	322	322	45.70	23.19	288	read
5	Norma	11	108	108	24.58	12.84	97	write
6	Norma	11	81	80	14.34	5.99	70	read
7	Norma	10	105	97	21.26	14.55	95	write
7	Saturn	11	139	95	25.55	12.05	128	write
8	Norma	10	132	68	21.30	15.26	122	write
8	Saturn	10	142	80	22.08	15.41	132	write
8	Moon	8	105	73	18.37	7.40	97	write
9	Norma	12	412	43	14.69	19.18	400	read
9	Saturn	12	86	79	14.99	6.36	74	read
9	Moon	10	75	11	11.99	4.76	65	read
10	Turtle1	~0	50	10	5.72	8.00	50	write
10	Sun58	34	345	307	48.37	31.19	311	write
11	Turtle1	~0	50	10	4.21	5.98	50	read
11	Sun58	32	241	229	37.57	18.38	209	read
12	Turtle1	~0	121	10	6.31	11.36	121	write
12	Turtle1	~0	3475	10	21.87	18.59	3475	write
12	Sun58	38	1680	316	69.90	82.61	1642	write
12	Sun58	35	2043	381	65.65	96.52	2008	write
13	Turtle1	~0	41	30	5.15	6.04	41	read
13	Turtle1	~0	5087	10	15.40	227.3	5087	read
13	Sun58	35	228	228	42.19	16.12	193	read
13	Sun58	34	1919	180	46.44	87.28	1885	read

Table 6.2 Data Statistics of Experiments

As seen in Table 7.2, we observed that for all experiments, first attempts for write/read services have a relatively longer response time than the average response time.

Second client on Turtle1 in experiment 13 might be thought as an exception for our observation. Note that this client had a worst-case response time of 5000 ms, and the response time of its first attempt would have been longer than the average if we excluded the worst-case data point from the average response time computation. Because of this, we did not conclude that this client was an exception for our observation. As mentioned in experiment 1, the reason is that JavaSpaces logs each contact in a file. This registration process makes first attempts for the services relatively longer. As a result of this observation, we implemented the network buffers so that when they are created, they write an entry to JavaSpaces to complete this registration process.

Another important observation was an upward trend in the response time diagrams for experiments 7 and 8. We conjectured that this might be the result of overhead introduced by memory allocation process in JavaSpaces write service when we have concurrent clients and high loads. The rationale for this conclusion was that we had no upward trend in experiment 9. Experiment 9 was similar to experiment 7 and 8 except that we tested the performance of JavaSpaces read service. Hence, data are promptly removed from the JavaSpaces by the destination network buffer in our implementation.

We observed that the read service has a relatively shorter response time than the write service. As mentioned, we conjectured that read service takes a sorter time than the write service because JavaSpaces must allocate a new memory space for each write service. As seen in Table 7.2, we had a few unexpectedly longer response times such as 1970 ms in experiment 2. We conjectured that this was the result of delay introduced by the operating system or overhead introduced by JavaSpaces. But we also had longer response times such as 2043 ms, 3475 ms, and 5087 ms., which all occurred in the experiments 12 and 13. The purpose of the experiments 12 and 13 was to test the performance of JavaSpaces under higher loads. Because of this we conjectured that these unexpectedly higher response times were the result of the higher load on the JavaSpaces instead of network congestion or delay introduced by operating system.

The Sun58 workstation has the slowest CPU (40 Mhz) among the machines we used in our experiments. So, we found that the highest response times were observed for clients running on this machine. The Sun58 increases the average response time for both

write/read services. The average response time we obtained for write service was 36.45 ms and 25.73 ms for read service. If we exclude the Sun58 response times, then we obtain an 18.56 ms average response time for write service and 11.53 ms for the read service. These values are very promising but the presence of unexpectedly high response times introduced by the unpredictable underlying network, delays introduced by operating system and overhead introduced by JavaSpaces prevent us from concluding that JavaSpaces are ideal for distributed real-time systems. As seen in the Table 7.2, the standard deviation of the response times is very high. Even though we have an average response time in tens of the milliseconds, these high standard deviations mean that the performance of JavaSpaces is highly unpredictable. Because predictability is an important requirement of real-time systems we cannot rely on JavaSpaces to provide that predictability. But, we can conclude that JavaSpaces has an average response time on the order of tens of the milliseconds and we can use JavaSpaces for distributed “soft” real-time systems, which require an average performance.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSION AND FUTURE WORKS

A. SUMMARY

This thesis proposes a proxy-based network buffer technique to be used as an inter-process communication layer and proposes a basic program structure for the Distributed Computer Aided Prototyping System (DCAPS) to automatically generate code for user-defined distributed real-time systems prototypes specified in PSDL for a given target platform.

The proxy-based network buffers provide the inter-process communication in a distributed heterogeneous environment. They act as if they are local buffers and shield the implementation and network operation details from the developers. They are built on Jini/JavaSpaces infrastructure to simplify the tasks of building and maintaining reliable distributed systems. We considered practical network issues such as transmission delay, loss of messages, and synchronization in the absence of a global clock. We have used JavaSpaces as a global clock to synchronize the distributed applications and as a repository for information exchange.

We have also found that any scheduling algorithm for a distributed real-time system must account for the CPU usage of inter-process communication operations. Inter-process communication takes relatively longer time than local operations. This relatively longer time consists of the time spent in the local CPU to initiate the network operations and the time spent in the network itself. This usage of CPU affects the scheduling of operators in the local CPU and becomes an important issue in finding a feasible schedule for a given distributed real-time system prototype specified in PSDL for a given target platform.

We have conducted several experiments to measure the response time of JavaSpaces. Results show that the latencies are in the order of milliseconds. We conjectured that without the support for real-time in underlying network and operating system, JavaSpaces are not sufficient to meet the requirements of a distributed real-time system because of the unpredictable nature of the network delays, delays introduced by

operating system and overhead introduced by JavaSpaces itself. We also applied a proxy-based network buffer technique to a sample prototype specified in PSDL for a given target platform. Our experiments showed that JavaSpaces provides an easy API to build distributed applications and it is only good for "soft" real-time systems because of its good average performance.

B. FUTURE WORK

This thesis provides an initial effort in research of using formal models and scheduling algorithms to build tools for automatically generating code for distributed real-time systems. We have provided a technique to be used in inter-process communications and investigated the effects of inter-process communications in the scheduling problem of distributed real-time systems. Improvements and additional research are needed in the following areas:

- Automatic Generation of Code:

  The primary benefit of any code generator is to reduce the amount of repetitive code that must be produced, thus saving time in the development cycle [RWJ01]. By using program structure proposed in this thesis, a code generator can be developed to automatically generate code for user-defined distributed real-time systems prototypes and integrated to the current DCAPS environment. Researches into the automatic generation code methods are recommended to improve current DCAPS.

- Partition of PSDL Graphs:

  It is difficult yet and important to decide which tasks will be executed on which processors in a distributed system. The partition of operators is an important issue for distributed real-time system prototypes specified in PSDL. An automated tool can be developed to partition PSDL graphs

across given target platforms. The output of this tool can then be passed to the generator to automatically generate the code for these prototypes. Scheduling algorithms for these tasks is another area for research. Static and dynamic scheduling algorithms can be researched and integrated in the current DCAPS.

- Modification to the current PSDL Editor:

  The current implementation of the PSDL Editor is not adequate for specifying distributed real-time systems. As we proposed in this thesis, the current PSDL model must be modified so that users may define different latencies for streams with the same identity but with different producers/consumers. The PSDL Editor must also automatically initialize unspecified latencies to default latency for each stream.

- Load Balancing:

  Load balancing is an important issue in building distributed systems. There are several different rationales and mechanisms employed for load balancing. Research into the relative strengths and weaknesses of these approaches are recommended to improve current DCAPS. Work done by Lap-Sun Cheung and Yu-Kwong Kwok [CAK01] offers a fuzzy decision based approach. In their work, they compare three different approaches, namely, JavaSpaces based, request redirection based, and fuzzy decision based approaches and conclude that the fuzzy decision based approach is the most promising one for load balancing.

C. CONCLUDING REMARKS

Our work demonstrates that we can use the JavaSpaces technology in the communication layer of the DCAPS for distributed soft real-time systems. We also argue

that JavaSpaces does not solve the global clock problem efficiently. The unpredictable latencies introduced by network make it difficult to synchronize the distributed applications using JavaSpaces. We also found and pointed out the deficiencies that must be improved for DCAPS (e.g., PSDL Editor modifications) to be a promising solution to the problems related to the development of distributed real-time system prototypes. The most emergent tool needed for current DCAPS is a distributed scheduler and translator.

APPENDIX A. JAVASPACES API

```
package net.jini.space;

public interface JavaSpace {

public final long NO_WAIT = 0; // do not wait

Lease write (Entry e, Transaction txn, long lease)
                                    throws RemoteException, TransactionException;

Entry read (Entry tmpl, Transaction txn, long timeout)
                                throws RemoteException,
                                    TransactionException,
                                    UnusableEntryException,
                                    InterruptedException;

Entry readIfExists (Entry tmpl, Transaction txn, long timeout)
                                throws RemoteException,
                                    TransactionException,
                                    UnusableEntryException,
                                    InterruptedException;

Entry take (Entry tmpl, Transaction txn, long timeout)
                                throws RemoteException,
                                    TransactionException,
                                    UnusableEntryException,
                                    InterruptedException;

Entry takeIfExists (Entry tmpl, Transaction txn, long timeout)
                                throws RemoteException,
                                    TransactionException,
```

```
                                UnusableEntryException,
                                InterruptedException;

 EntryRegistration notify (Entry tmpl,
                           Transaction txn,
                           RemoteEventListener l,
                           long lease,
                           MarshalledObject obj)
                                throws RemoteException, TransactionException;

  Entry snapshot (Entry e) throws RemoteException;
} // end of JavaSpace interface
```

APPENDIX B. IMPLEMENTATION CODE

A. COMPUTATION UNIT ONE

```
/**
 * Title:        ProcessorOne class
 * Description:  This class is the implementation of applicatin part
 *               for processor one of a distributed application
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */
package dcapsone;

import spacediscovery.Discovery;
import StartEntry;
import NetworkDoubleSampledBuffer;
import NetworkDoubleFIFOBuffer;
import localbuffer.*;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
public class ProcessorOne {
  // JavaSpace for network communications
  private JavaSpace javaSpace;

  // network stream buffers
  private NetworkDoubleSampledBuffer temperature_GUI;
  private NetworkDoubleSampledBuffer valve_adjustment_GUI;
  private NetworkDoubleSampledBuffer fuel_GUI;
  private NetworkDoubleFIFOBuffer valve_adjustment_Valve_Control;

  // network state streams
  private NetworkDoubleSampledBuffer valve_state_Monitor_Environment;

  // local state stream buffers
  private DoubleStateStreamBuffer valve_state_GUI;
  private DoubleStateStreamBuffer valve_state_Valve_Control;

  // Operator instances
  private Valve_Control valve_control;
  private GUI gui;

  // period of harmonic block
  private long period;

  // stream variables for GUI operator
  private Double temperatureGUI;
  private Double valve_adjustmentGUI;
  private Double fuelGUI;
  private Double valve_stateGUI;
```

```
// stream variables for Valve_Control operator
private Double valve_adjustmentVC;
private Double valve_stateVC;

// timestamp for harmonic block
private long beginning;
private long last;


// inner class to listen start notification
class Listener extends UnicastRemoteObject implements RemoteEventListener {
 /*
 * Default constructor
 */
 public Listener() throws RemoteException {
 } // end of default constructor


 /*
 *  This method is notified by the JavaSpaces when a new message is written
 */
 public void notify(RemoteEvent ev) {
   //System.out.println("Starting new period");
   // we got the start notification, start the application driver thread
   start();
 } // end of notify method
} // end of inner class Listener
```

```
  // operator implementations as inner classes
  class Valve_Control {//implements Runnable {
    public Valve_Control() {
    } // end of inner class Temperature_Control constructor

    private void valve_control() {
double state = valve_stateVC.doubleValue() +   valve_adjustmentVC.doubleValue();

      if (state < 0) {
        valve_stateVC = new Double(0);
      } else if (state > 1) {
        valve_stateVC = new Double(1);
      } else {
        valve_stateVC = new Double(state);
      } // end of if else
    } // end of valve_control
  } // end of inner class Temperature_Control


  class GUI extends JFrame {
    // GUI elements
    JLabel space;
    JLabel info;
    JLabel temp_label;
    JLabel valve_adjust_label;
    JLabel fuel_label;
    JLabel valve_state_label;
    Font f = new Font("", 1, 16);
    Color red = Color.red;
    Color green = Color.green;
    Color blue = Color.blue;
```

```
public GUI() {
  super("Monitor");
  setSize(500, 500);
  setLocation(200, 200);
  setResizable(false);
  Container c = this.getContentPane();
  c.setLayout(new GridLayout(6, 1));

  if (javaSpace != null) {
     space = new JLabel("Connected");
     space.setForeground(blue);
  }
  else {
     space = new JLabel("Disconnected");
     space.setForeground(red);
  } // end of if else

  info = new JLabel(System.getProperty("java.rmi.server.codebase"));
  info.setForeground(blue);
  temp_label = new JLabel("not available");
  temp_label.setForeground(green);
  valve_adjust_label = new JLabel("not available");
  valve_adjust_label.setForeground(green);
  fuel_label = new JLabel("not available");
  fuel_label.setForeground(green);
  valve_state_label = new JLabel("not available");
  valve_state_label.setForeground(green);

  // setting fonts
  space.setFont(f);
  info.setFont(f);
  temp_label.setFont(f);
```

```
        valve_adjust_label.setFont(f);
        fuel_label.setFont(f);
        valve_state_label.setFont(f);

        // setting color and borders
        space.setBorder(BorderFactory.createTitledBorder("JavaSpace Status"));
        info.setBorder(BorderFactory.createTitledBorder("JavaSpace Server Codebase"));
        temp_label.setBorder(BorderFactory.createTitledBorder("Current Temperature
(F)"));
        valve_adjust_label.setBorder(BorderFactory.createTitledBorder("Current Valve
Adjustment (%)"));
        fuel_label.setBorder(BorderFactory.createTitledBorder("Remaining Fuel
(gallons)"));
        valve_state_label.setBorder(BorderFactory.createTitledBorder("Current Valve
State"));

        // adding labels
        c.add(space);
        c.add(info);
        c.add(temp_label);
        c.add(valve_adjust_label);
        c.add(fuel_label);
        c.add(valve_state_label);

        // window listener
        addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
            System.exit(0);
          } // end of windowClosing
        });

        setVisible(true);
      } // end of inner class GUI constructor
```

```
public void gui() {
   // updating GUI
   if (javaSpace == null) {
     space.setText("Disconnected");
     space.setForeground(red);
   } // end of if

   if (temperatureGUI == null) {
     //temp_label.setText("not available");
     temp_label.setForeground(green);
   }
   else {
     temp_label.setText(temperatureGUI.toString());
     if (temperatureGUI.doubleValue() < 70 || temperatureGUI.doubleValue() > 80) {
       temp_label.setForeground(red);
     } else {
       temp_label.setForeground(blue);
     } // end of if else
   } // end of if else


   if (valve_adjustmentGUI == null) {
     //valve_adjust_label.setText("not available");
     valve_adjust_label.setForeground(green);
   }
   else {
     valve_adjust_label.setText(valve_adjustmentGUI.toString());
     valve_adjust_label.setForeground(blue);
   } // end of if else
```

```
        if (fuelGUI == null) {
          //fuel_label.setText("not available");
          fuel_label.setForeground(green);
        }
        else {
          if (fuelGUI.doubleValue() <= 0) {
            fuel_label.setText("NO FUEL, ENVIRONMENT CONTROL IS STOPPED");
            fuel_label.setForeground(red);
           } else if (fuelGUI.doubleValue() <= 0.3) {
             fuel_label.setText(fuelGUI.toString() + " WARNING: FUEL LEVEL IS TOO
LOW");
            fuel_label.setForeground(Color.yellow);
          } else {
            fuel_label.setText(fuelGUI.toString());
            fuel_label.setForeground(blue);
          } // end of if else
        } // end of if else

        if (valve_stateGUI == null) {
          //valve_state_label.setText("not available");
          valve_state_label.setForeground(green);
        }
        else {
          valve_state_label.setText(valve_stateGUI.toString());
          valve_state_label.setForeground(blue);
        } // end of if else
      } // end of gui
    } // end of inner class GUI
```

```
// driver method, it is called when we got the start message from the main controller
  public void start() {
    new Thread(new Runnable() {
      public void run() {
       try {
          beginning = System.currentTimeMillis();

          // sleep for the first 1700 ms
          Thread.sleep(1700);
          // check for newData
          if (valve_adjustment_Valve_Control.newData()) {
             // time to fire valve control time critical op
             // reading from streams
             valve_adjustmentVC = new Double(valve_adjustment_Valve_Control.read());
             valve_stateVC      = new Double(valve_state_Valve_Control.read());

             // firing operator
             valve_control.valve_control();

             // writing to the local streams
             valve_state_GUI.write(valve_stateVC.doubleValue());
             valve_state_Valve_Control.write(valve_stateVC.doubleValue());

             // check for period time
             if (( last = 1800 - (System.currentTimeMillis() - beginning)) < 0) {
                // timing error
                last = 0;
                System.out.println("Timing error in Valve_Control operator firing");
             } // end of it
             // writing to the netwok streams
             valve_state_Monitor_Environment.write(valve_stateVC.doubleValue());
          } // end of if
```

```
  // sleep for max. latency
  Thread.sleep(last);

  // we can now fire the GUI non time critical operator
  // reading variables for GUI
  temperatureGUI       = new Double(temperature_GUI.read());

  if (valve_adjustment_GUI.newData()) {
    valve_adjustmentGUI  = new Double(valve_adjustment_GUI.read());
  } else {
    valve_adjustment_GUI = null;
  }

  if (fuel_GUI.newData()) {
    fuelGUI              = new Double(fuel_GUI.read());
  } else {
    fuelGUI              = null;
  }

  valve_stateGUI       = new Double(valve_state_GUI.read());
  gui.gui();

  if ((last = period - (System.currentTimeMillis() - beginning)) < 0) {
  // timing error
     last = 0;
     System.out.println("Timing error in GUI operator");
  } // end of if;

  // sleep for the remaining of the period
  Thread.sleep(last);
}
```

```
      catch (InterruptedException e) {
        e.printStackTrace();
      } // end of try catch
    }
   }).start();
  } // end of start


  // constructor
  public ProcessorOne() {
    // initialization of JavaSpace
    if ((javaSpace =
Discovery.discoverJavaSpace(getJiniScheme(System.getProperty("java.rmi.server.codeb
ase")))) == null) {
      // no java space, warn the user and exit
      System.out.println("Couldn't find a JavaSpace! Exiting...");
      System.exit(0);
    }
    else {
     // initializations of network buffers
      temperature_GUI      = new NetworkDoubleSampledBuffer("temperature_GUI",
javaSpace, 1100);
      valve_adjustment_GUI = new
NetworkDoubleSampledBuffer("valve_adjustment_GUI", javaSpace, 1100);
      fuel_GUI = new NetworkDoubleSampledBuffer("fuel_GUI", javaSpace, 1100);
      valve_adjustment_Valve_Control = new
NetworkDoubleFIFOBuffer("valve_adjustment_Valve_Control", javaSpace, 1000);



      // initializations of state streams
      valve_state_GUI = new DoubleStateStreamBuffer(0);
      valve_state_Valve_Control = new DoubleStateStreamBuffer(0);
      valve_state_Monitor_Environment = new NetworkDoubleSampledBuffer(0,
"valve_state_Monitor_Environment", javaSpace, 1100);
```

```
        // activation of network streams which will be used to read
        temperature_GUI.setNotification();
        valve_adjustment_Valve_Control.setNotification();
        fuel_GUI.setNotification();
        valve_adjustment_GUI.setNotification();

        // initializations of operator instances
        valve_control = new Valve_Control();
        gui           = new GUI();

        // period of harmonc block
        period = 3000;

        // registration to listen for start notification
        try {
           javaSpace.notify(new StartEntry(), null, new Listener(), Long.MAX_VALUE,
null);
           System.out.println("Waiting for start notification...");
        }
        catch (RemoteException e) {
          System.out.println("There is a problem with remote object, exiting");
          e.printStackTrace();
          System.exit(0);
        }
        catch (Exception e) {
          System.out.println("Unexpected exception, exiting");
          e.printStackTrace();
          System.exit(0);
        } // end of try catch
      } // end of if else
    } // end of ProcessorOne constructor
```

```
  private String getJiniScheme(String s) {
    int last = s.lastIndexOf(':');
    return "jini://" + s.substring(7, last);
  } // end of getJiniScheme

  public static void main(String[] args) {
    // setting security file
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    } // end of if
    ProcessorOne p = new ProcessorOne();
  } // end of main
} // end of ProcessorOne class
```

THIS PAGE INTENTIONALLY LEFT BLANK

B. COMPUTATION UNIT TWO

```
/**
 * Title:        ProcessorTwo class
 * Description:  This class is the implementation of applicatin part
 *               for processor two of a distributed application
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */
package dcapstwo;

import spacediscovery.Discovery;
import StartEntry;
import NetworkDoubleFIFOBuffer;
import NetworkDoubleSampledBuffer;
import localbuffer.*;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;


public class ProcessorTwo {
  // JavaSpace for network communications
  private JavaSpace javaSpace;

  // local stream buffers
  private LocalDoubleSampledBuffer temperature_Temperature_Control;
```

```
// network stream buffers
private NetworkDoubleSampledBuffer temperature_GUI;
private NetworkDoubleSampledBuffer valve_adjustment_GUI;
private NetworkDoubleSampledBuffer fuel_GUI;
private NetworkDoubleFIFOBuffer valve_adjustment_Valve_Control;

// network state streams
private NetworkDoubleSampledBuffer valve_state_Monitor_Environment;

// local state streams
private DoubleStateStreamBuffer fuel_Temperature_Control;
private DoubleStateStreamBuffer temperature_Monitor_Environment;

// Operator instances
Temperature_Control temperature_control;
Monitor_Environment monitor_environment;

// period of harmonic block
private long period;

// stream variables for temperature_control operator
private Double temperatureTC;
private Double valve_adjustmentTC;
private Double fuelTC;

// stream variables for monitor environment operator
private Double temperatureME;
private Double valve_stateME;

// timestamp for harmonic block
private long beginning;
```

```
  private long last;
  // inner class to listen start notification
  class Listener extends UnicastRemoteObject implements RemoteEventListener {
   /*
   * Default constructor
   */
    public Listener() throws RemoteException {
   } // end of default constructor



   /*
    * This method is notified by the JavaSpaces when a new message is written into the
JavaSpaces
   */
    public void notify(RemoteEvent ev) {
      //System.out.println("Starting new period");
      // we got the start notification, start the application
      start();
   } // end of notify method
  } // end of inner class Listener



  // operator implementations as inner classes
  class Temperature_Control {


     public Temperature_Control() {
    } // end of inner class Temperature_Control constructor
```

```
    private void temperature_control() {
      double fu = fuelTC.doubleValue();

      if (fu > 0) {
        if (temperatureTC.doubleValue() < 70) {
          valve_adjustmentTC = new Double(-0.1);
          fuelTC = new Double(fu - 0.0001);
        } else if (temperatureTC.doubleValue() > 80) {
          valve_adjustmentTC = new Double(0.1);
          fuelTC = new Double(fu - 0.0001);
        } else {
          valve_adjustmentTC = new Double(0.00099);
        } // end of if else
      } else {
        // do nothing
        //no fuel
      } // end of if else
    } // end of temperature_control
  } // end of inner class Temperature_Control

  class Monitor_Environment {

    public Monitor_Environment() {
    } // end of inner class Monitor_Environment constructor


    private void monitor_environment() {
      temperatureME = new Double(temperatureME.doubleValue() + 0.1 -
valve_stateME.doubleValue());
    } // end of monitor_environment
  } // end of inner class Monitor_Environment
```

```
// driver method, it is called when we got the start message from the main controller
public void start() {
  new Thread(new Runnable() {
    public void run() {
      try {
        beginning = System.currentTimeMillis();

        // reading variables for Monitor_Environment
        valve_stateME = new Double(valve_state_Monitor_Environment.read());
        temperatureME = new Double(temperature_Monitor_Environment.read());

        // firing Monitor_Environment operator
        monitor_environment.monitor_environment();

        // writing to local streams
        temperature_Monitor_Environment.write(temperatureME.doubleValue());
        temperature_Temperature_Control.write(temperatureME.doubleValue());

        if ((last = 100 - (System.currentTimeMillis() - beginning)) < 0) {
          // timing error
          last = 0;
          System.out.println("Timing error in monitor environment operator");
        } // end of if

        // writing to network streams
        temperature_GUI.write(temperatureME.doubleValue());

        Thread.sleep(last);

        if (temperature_Temperature_Control.newData()) {
          // time for firing of temperature control operator
```

```
            // reading from streams
            temperatureTC = new Double(temperature_Temperature_Control.read());
            fuelTC = new Double(fuel_Temperature_Control.read());

            if (temperatureTC == null || fuelTC == null) {
              // reading error
              System.out.println("Stream reading error in temperature control operator");
            } // end of if

            // firing operator
            temperature_control.temperature_control();

            // writing to the local stream
            fuel_Temperature_Control.write(fuelTC.doubleValue());

            // time check
            if ((System.currentTimeMillis() - beginning) > 700) {
               // timing error
               System.out.println("Timing error in temperature control operator");
            } // end of if

            // writing to the network streams
            if (valve_adjustmentTC.doubleValue() > 0.01 ||
valve_adjustmentTC.doubleValue() < - 0.01) {

valve_adjustment_Valve_Control.write(valve_adjustmentTC.doubleValue());
            } // end of if
            valve_adjustment_GUI.write(valve_adjustmentTC.doubleValue());
            fuel_GUI.write(fuelTC.doubleValue());
          } // end of if
```

```
            // time check
            if ((last = period - (System.currentTimeMillis() - beginning)) < 0) {
               // timing error
               last = 0;
               System.out.println("Timing error in temperature control operator");
            } // end of if

            // sleep for the rest of the period
            Thread.sleep(last);
         }
         catch (InterruptedException e) {
            e.printStackTrace();
         } // end of try catch
      }
    }).start();
  } // end of start



  public ProcessorTwo() {
     // initialization of JavaSpace
     if ((javaSpace =
Discovery.discoverJavaSpace(getJiniScheme(System.getProperty("java.rmi.server.codeb
ase")))) == null) {
        // no java space, warn the user and exit
        System.out.println("Couldn't find a JavaSpace! Exiting...");
        System.exit(0);
     }
     else {
        // initializations of streams
        temperature_Temperature_Control = new LocalDoubleSampledBuffer();
```

```
        temperature_GUI      = new NetworkDoubleSampledBuffer("temperature_GUI",
javaSpace, 1100);
        valve_adjustment_GUI = new
NetworkDoubleSampledBuffer("valve_adjustment_GUI", javaSpace, 1100);
        fuel_GUI = new NetworkDoubleSampledBuffer("fuel_GUI", javaSpace, 1100);
        valve_adjustment_Valve_Control = new
NetworkDoubleFIFOBuffer("valve_adjustment_Valve_Control", javaSpace, 1000);
        valve_state_Monitor_Environment = new NetworkDoubleSampledBuffer(0,
"valve_state_Monitor_Environment", javaSpace, 1100);


        fuel_Temperature_Control = new DoubleStateStreamBuffer(1);
        temperature_Monitor_Environment = new DoubleStateStreamBuffer(75);


        // activation of network streams which will be used to read
        valve_state_Monitor_Environment.setNotification();


        // initializations of operator instances
        temperature_control = new Temperature_Control();
        monitor_environment = new Monitor_Environment();


        // period of harmonic block
        period = 3000;
        // registration to listen  for start notification
        try {
           javaSpace.notify(new StartEntry(), null, new Listener(), Long.MAX_VALUE,
null);
           System.out.println("Waiting for start notification...");
        }
        catch (RemoteException e) {
          System.out.println("There is a problem with remote object, exiting");
          e.printStackTrace();
          System.exit(0);
        }
```

```
  catch (Exception e) {
          System.out.println("Unexpected exception, exiting");
          e.printStackTrace();
          System.exit(0);
        } // end of try catch
    } // end of if else
  } // end of constructor



  private String getJiniScheme(String s) {
    int last = s.lastIndexOf(':');
    return "jini://" + s.substring(7, last);
  } // end of getJiniScheme



  public static void main(String[] args) {
    // setting security file
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    } // end of if

    ProcessorTwo p = new ProcessorTwo();
  } // end of main
} // end of ProcessorTwo class
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. MASTER APPLICATION

```
/**
 * Title:        Starter class
 * Description:  To start distributed application by placing a start entry
 *               into the JavaSpaces
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */
package starter;


import java.rmi.RMISecurityManager;
import net.jini.space.JavaSpace;
import spacediscovery.Discovery;
import StartEntry;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Starter extends JFrame {

  private String address;
  private JLabel labelOne;
  private JLabel labelTwo;
  private JLabel labelThree;
  private JLabel labelFour;
  private JTextField fieldOne;
  private JTextField fieldTwo;
  private JTextField fieldThree;
```

```
private JTextField fieldFour;
private JLabel status;
private JButton start = new JButton("Start");
private long period;
private StartEntry entry;

public Starter() {
  super("Start");
  setSize(450, 250);
  setLocation(300, 150);
  setResizable(false);

  entry    = new StartEntry();
  labelOne = new JLabel(" JavaSpace Server Codebase");
  labelOne.setBorder(BorderFactory.createEtchedBorder());
  labelOne.setForeground(Color.black);
  labelTwo = new JLabel(" Security File");
  labelTwo.setForeground(Color.black);
  labelTwo.setBorder(BorderFactory.createEtchedBorder());
  labelThree = new JLabel(" LCM (miliseconds)");
  labelThree.setForeground(Color.black);
  labelThree.setBorder(BorderFactory.createEtchedBorder());
  labelFour = new JLabel(" Status");
  labelFour.setForeground(Color.black);
  labelFour.setBorder(BorderFactory.createEtchedBorder());
  fieldFour = new JTextField("Waiting for start command...");
  start = new JButton("Start Distributed Application");

  fieldOne   = new JTextField("http://131.120.8.41:8081/");
  fieldTwo   = new JTextField("c:\\policy.all");
  fieldThree = new JTextField("500");
```

```
    start.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        startApplication();
      }
    });
    Container c = getContentPane();
    c.setLayout(new GridLayout(5, 1));
    JPanel p1 = new JPanel(new GridLayout(1, 2));
    JPanel p2 = new JPanel(new GridLayout(1, 2));
    JPanel p3 = new JPanel(new GridLayout(1, 2));
    JPanel p4 = new JPanel(new GridLayout(1, 2));
    p1.add(labelOne);
    p1.add(fieldOne);
    p2.add(labelTwo);
    p2.add(fieldTwo);
    p3.add(labelThree);
    p3.add(fieldThree);
    p4.add(labelFour);
    p4.add(fieldFour);

    c.add(p1);
    c.add(p2);
    c.add(p3);
    c.add(p4);
    c.add(start);
    setVisible(true);
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
  } // end of default constructor
```

```
private void startApplication() {
  fieldFour.setText("Trying to connect to the JavaSpace...");
  address = fieldOne.getText();

  try {
    period  = Long.valueOf(fieldThree.getText()).longValue();
  }
  catch (NumberFormatException e) {
    e.printStackTrace();
  } // end of try catch


  System.setProperty("java.rmi.server.codebase", address);
  System.setProperty("java.security.policy", fieldTwo.getText());

  final JavaSpace space = Discovery.discoverJavaSpace(getJiniScheme(address));
  if (space == null) {
    fieldFour.setText("Couldn't connect to the JavaSpace...");
  }
  else {
    fieldFour.setText("Connected to the JavaSpace...");
    fieldFour.setText("Trying to start distributed application...");
    new Thread(new Runnable() {

      public void run() {

        try {
          while(true) {
          space.write(entry, null, 50000);
          Thread.sleep(period);
        } // end of while
      }
```

```
            catch (Exception e) {
              e.printStackTrace();
            } // end of try catch
          } // end of run
        }).start();

        fieldFour.setText("Distributed application was started...");
      } // end of if else
    } // end of startApplication



    private String getJiniScheme(String s) {
      int last = s.lastIndexOf(':');
      return "jini://" + s.substring(7, last);
    } // end of getJiniScheme



    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
         System.setSecurityManager(new RMISecurityManager());
      }

      Starter s = new Starter();
    } // end of main
  } // end of Starter class
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. JAVASPACES DISCOVERY CLASS

```
/**
 * Title:        Discovery class
 * Description:  This class discovers a local jini lookup service
 *               and registers to a JavaSpace
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */
package spacediscovery;

import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.space.JavaSpace;
import java.io.*;
import java.rmi.RemoteException;
import java.net.MalformedURLException;

public class Discovery {

   private static JavaSpace javaSpace;
   private static Class[] types = {JavaSpace.class};
   private static ServiceTemplate tmpl = new ServiceTemplate(null, types, null);
```

```
// This method finds the registrar of a lookup service
private static ServiceRegistrar getRegistrar(String url)
    throws MalformedURLException, IOException, ClassNotFoundException {
   LookupLocator loc = new LookupLocator(url);
   return loc.getRegistrar();
} // end of getRegistrar

// this method finds the JavaSpaces
private static void findJavaSpace(ServiceRegistrar reg) {
   if (javaSpace != null) {
      return;
   }
   else {
      try {
         javaSpace = (JavaSpace) reg.lookup(tmpl);

         if (javaSpace == null) {
            return;
         } // end of if
      }
      catch (RemoteException e) {
         e.printStackTrace();
      } // end of try catch
   } // end of if else
} // end of findJavaSpace
```

```
  // This method discovers and returns a JavaSpace. If it cannot find a
  // local JavaSpace returns a null reference
  public static JavaSpace discoverJavaSpace(String url) {

    try {
      findJavaSpace(getRegistrar(url));
    }
    catch (Exception e) {
      e.printStackTrace();
    } // end of try catch

    return javaSpace;
  } // end ofdiscoverJavaSpace
} // end of Discovery class
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E. NETWORK BUFFERS

A. SAMPLED STREAM NETWORK BUFFERS

1. String Type

```
/**
 * Title:        NetworkStringSampledBuffer
 * Description:  This class implements a Sampled buffer using JavaSpace for String values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import tuplespace.entries.*;
```

```
public class NetworkStringSampledBuffer {

  private JavaSpace space;
  private EntryString variable;
  private Entry tempOne;
  private Entry tempTwo;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;



  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements
RemoteEventListener {

    public EvtListener() throws RemoteException {
    } // end of EvtListener default constructor



    public void notify(RemoteEvent ev) {
      readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
        /*
        *  Constructor
        */
        public NetworkStringSampledBuffer(String id, JavaSpace space, long latency) {
          this.space    = space;
          this.id       = id;
          this.latency  = latency;
          try {
             this.template = this.space.snapshot(new EntryString(id));
             this.space.write(new EntryString(), null, 6000);
          }
          catch (Exception e) {
            e.printStackTrace();
          }
          this.newData  = false;
        } // end of constructor


        /*
        *  Constructor for state stream mode
        */
        public NetworkStringSampledBuffer(String x, String id, JavaSpace space, long
latency) {
           this.variable = new EntryString(id, x);
          this.space    = space;
          this.id       = id;
          this.latency  = latency;

          try {
             this.template = this.space.snapshot(new EntryString(id));
             this.space.write(new EntryString(), null, 6000);
          }
```

```
    catch (Exception e) {
      e.printStackTrace();
    }
    this.newData  = false;
  } // end of constructor
  ////////////////////////////////////////////////////////////////////////
  // public methods
  ////////////////////////////////////////////////////////////////////////
  /*
  * This method returns the current value
  */
  synchronized public String read() {
    newData = false;
    return variable.entryString.toString();
  } // end of read
 /*
  * This method writes the given value to the javaspace
  */
  synchronized public void write(String value) {
    EntryString entry = new EntryString(id, value);
    if (space != null) {
      try {
         space.write(entry, null, latency);
      }
      catch (RemoteException ex) {
        ex.printStackTrace();
      }
      catch (Exception ex) {
        ex.printStackTrace();
      } // end of try-catch statement
    } // end of if statement
  } // end of write
```

```
/*
* This method returns the newData value
*/
synchronized public boolean newData() {
   return newData;
} // end of newData



 /*
  * This method is used to enable notification
  */
  public void setNotification() {
   try {
      EvtListener listener = new EvtListener();
      if (space != null) {
        space.notify(template, null, listener, Long.MAX_VALUE, null);
      } // end of if statement
   }
   catch (Exception ex) {
      ex.printStackTrace();
   } // end of try-catch statement
  } // end of setNotification
```

```
    ////////////////////////////////////////////////////////////////////////
    // private methods
    ////////////////////////////////////////////////////////////////////////
    /*
     * This method is called by the notify method of EvtListener object
     * When called, it takes every object in the javaspace with predefined id
     */
    private void readFromSpace() {
      try {
        while((tempOne = space.takeIfExists(template, null,
JavaSpace.NO_WAIT)) != null) {
          tempTwo = tempOne;
        };

        if (tempTwo != null) {
          variable = (EntryString) tempTwo;
          newData  = true;
        } // end of if statement
      }
      catch (Exception ex) {
        ex.printStackTrace();
      } // end of try-catch statement
    } // end of readFromSpace
  } // end of NetworkStringSampledBuffer class
```

2. Boolean Type

```
/**
 * Title:        NetworkBooleanSampledBuffer
 * Description:  This class implements a Sampled buffer using JavaSpace for Boolean values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import  tuplespace.entries.*;
```

```
public class NetworkBooleanSampledBuffer {

  private JavaSpace space;
  private EntryBoolean variable;
  private Entry tempOne;
  private Entry tempTwo;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;



  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

    public EvtListener() throws RemoteException {
    } // end of EvtListener default constructor


    public void notify(RemoteEvent ev) {
      readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
 public NetworkBooleanSampledBuffer(String id, JavaSpace space, long latency) {
   this.space    = space;
   this.id       = id;
   this.latency  = latency;
   try {
      this.template = this.space.snapshot(new EntryBoolean(id));
      this.space.write(new EntryBoolean(), null, 6000);
   }
   catch (Exception e) {
     e.printStackTrace();
   }
   this.newData  = false;
 } // end of constructor



 /*
 *  Constructor for state stream mode
 */
 public NetworkBooleanSampledBuffer(boolean x, String id, JavaSpace space, long
latency) {
   this.variable = new EntryBoolean(id, x);
   this.space    = space;
   this.id       = id;
   this.latency  = latency;


   try {
      this.template = this.space.snapshot(new EntryBoolean(id));
      this.space.write(new EntryBoolean(), null, 6000);
   }
```

```
    catch (Exception e) {
      e.printStackTrace();
    }
    this.newData  = false;
  } // end of constructor
  ////////////////////////////////////////////////////////////////////////
  // public methods
  ////////////////////////////////////////////////////////////////////////
  /*
  * This method returns the current value
  */
  synchronized public boolean read() {
    newData = false;
    return variable.entryBoolean.booleanValue();
  } // end of read
/*
  * This method writes the given value to the javaspace
  */
  synchronized public void write(boolean value) {
    EntryBoolean entry = new EntryBoolean(id, value);
    if (space != null) {
      try {
         space.write(entry, null, latency);
      }
      catch (RemoteException ex) {
        ex.printStackTrace();
      }
      catch (Exception ex) {
        ex.printStackTrace();
      } // end of try-catch statement
    } // end of if statement
  } // end of write
```

```
/*
* This method returns the newData value
*/
synchronized public boolean newData() {
    return newData;
} // end of newData



 /*
 * This method is used to enable notification
 */
 public void setNotification() {
  try {
     EvtListener listener = new EvtListener();
     if (space != null) {
        space.notify(template, null, listener, Long.MAX_VALUE, null);
     } // end of if statement
  }
  catch (Exception ex) {
     ex.printStackTrace();
  } // end of try-catch statement
 } // end of setNotification
```

```
  ////////////////////////////////////////////////////////////////////////
  // private methods
  ////////////////////////////////////////////////////////////////////////

  /*
  * This method is called by the notify method of EvtListener object
  * When called, it takes every object in the javaspace with predefined id
  */
  private void readFromSpace() {
    try {
      while((tempOne = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) !=
null) {
        tempTwo = tempOne;
      };

      if (tempTwo != null) {
        variable = (EntryBoolean) tempTwo;
        newData  = true;
      } // end of if statement
    }
    catch (Exception ex) {
      ex.printStackTrace();
    } // end of try-catch statement
  } // end of readFromSpace
} // end of NetworkBooleanSampledBuffer class
```

3. Integer Type

```
/**
 * Title:        NetworkIntegerSampledBuffer
 * Description:  This class implements a Sampled buffer using JavaSpace for Integer values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import tuplespace.entries.*;
```

```
public class NetworkIntegerSampledBuffer {

  private JavaSpace space;
  private EntryInteger variable;
  private Entry tempOne;
  private Entry tempTwo;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;


  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

    public EvtListener() throws RemoteException {
    } // end of EvtListener default constructor


    public void notify(RemoteEvent ev) {
      readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
public NetworkIntegerSampledBuffer(String id, JavaSpace space, long latency) {
  this.space    = space;
  this.id       = id;
  this.latency  = latency;
  try {
     this.template = this.space.snapshot(new EntryInteger(id));
     this.space.write(new EntryInteger(), null, 6000);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  this.newData  = false;
} // end of constructor


/*
*  Constructor for state stream mode
*/
public NetworkIntegerSampledBuffer(int x, String id, JavaSpace space, long latency) {
   this.variable = new EntryInteger(id, x);
  this.space    = space;
  this.id       = id;
  this.latency  = latency;

  try {
     this.template = this.space.snapshot(new EntryInteger(id));
     this.space.write(new EntryInteger(), null, 6000);
  }
```

```
    catch (Exception e) {
      e.printStackTrace();
    }
    this.newData  = false;
  } // end of constructor
  ////////////////////////////////////////////////////////////////////////
  // public methods
  ////////////////////////////////////////////////////////////////////////
  /*
  * This method returns the current value
  */
  synchronized public int read() {
    newData = false;
    return variable.entryInteger.intValue();
  } // end of read
 /*
  * This method writes the given value to the javaspace
  */
  synchronized public void write(int value) {
    EntryInteger entry = new EntryInteger(id, value);
    if (space != null) {
      try {
        space.write(entry, null, latency);
      }
      catch (RemoteException ex) {
        ex.printStackTrace();
      }
      catch (Exception ex) {
        ex.printStackTrace();
      } // end of try-catch statement
    } // end of if statement
  } // end of write
```

```
/*
* This method returns the newData value
*/
synchronized public boolean newData() {
    return newData;
} // end of newData



 /*
 * This method is used to enable notification
 */
 public void setNotification() {
  try {
     EvtListener listener = new EvtListener();
     if (space != null) {
        space.notify(template, null, listener, Long.MAX_VALUE, null);
     } // end of if statement
  }
  catch (Exception ex) {
     ex.printStackTrace();
  } // end of try-catch statement
 } // end of setNotification
```

```
  ////////////////////////////////////////////////////////////////////////
  // private methods
  ////////////////////////////////////////////////////////////////////////

  /*
  * This method is called by the notify method of EvtListener object
  * When called, it takes every object in the javaspace with predefined id
  */
  private void readFromSpace() {
    try {
      while((tempOne = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) !=
null) {
        tempTwo = tempOne;
      };

      if (tempTwo != null) {
        variable = (EntryInteger) tempTwo;
        newData  = true;
      } // end of if statement
    }
    catch (Exception ex) {
      ex.printStackTrace();
    } // end of try-catch statement
  } // end of readFromSpace
} // end of NetworkIntegerSampledBuffer class
```

4. Double Type

```
/**
 * Title:        NetworkDoubleSampledBuffer
 * Description:  This class implements a Sampled buffer using JavaSpace for double values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import tuplespace.entries.*;
```

```
public class NetworkDoubleSampledBuffer {

  private JavaSpace space;
  private EntryDouble variable;
  private Entry tempOne;
  private Entry tempTwo;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;


  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

    public EvtListener() throws RemoteException {
    } // end of EvtListener default constructor


    public void notify(RemoteEvent ev) {
      readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
 public NetworkDoubleSampledBuffer(String id, JavaSpace space, long latency) {
   this.space    = space;
   this.id       = id;
   this.latency  = latency;
   try {
      this.template = this.space.snapshot(new EntryDouble(id));
      this.space.write(new EntryDouble(), null, 6000);
   }
   catch (Exception e) {
    e.printStackTrace();
   }
   this.newData  = false;
 } // end of constructor



 /*
 *  Constructor for state stream mode
 */
  public NetworkDoubleSampledBuffer(double x, String id, JavaSpace space, long
latency) {
    this.variable = new EntryDouble(id, x);
   this.space    = space;
   this.id       = id;
   this.latency  = latency;


   try {
      this.template = this.space.snapshot(new EntryDouble(id));
      this.space.write(new EntryDouble(), null, 6000);
   }
```

```
    catch (Exception e) {
      e.printStackTrace();
    }
    this.newData  = false;
  } // end of constructor
  ////////////////////////////////////////////////////////////////////////
  // public methods
  ////////////////////////////////////////////////////////////////////////
  /*
  * This method returns the current value
  */
  synchronized public double read() {
    newData = false;
    return variable.entryDouble.doubleValue();
  } // end of read
 /*
  * This method writes the given value to the javaspace
  */
  synchronized public void write(double value) {
    EntryDouble entry = new EntryDouble(id, value);
    if (space != null) {
      try {
         space.write(entry, null, latency);
      }
      catch (RemoteException ex) {
        ex.printStackTrace();
      }
      catch (Exception ex) {
        ex.printStackTrace();
      } // end of try-catch statement
    } // end of if statement
   } // end of write
```

```
/*
* This method returns the newData value
*/
synchronized public boolean newData() {
    return newData;
} // end of newData



 /*
 * This method is used to enable notification
 */
 public void setNotification() {
  try {
     EvtListener listener = new EvtListener();
     if (space != null) {
        space.notify(template, null, listener, Long.MAX_VALUE, null);
     } // end of if statement
  }
  catch (Exception ex) {
     ex.printStackTrace();
  } // end of try-catch statement
 } // end of setNotification
```

```
  ////////////////////////////////////////////////////////////////////////
  // private methods
  ////////////////////////////////////////////////////////////////////////



  /*
  * This method is called by the notify method of EvtListener object
  * When called, it takes every object in the javaspace with predefined id
  */
  private void readFromSpace() {
    try {
      while((tempOne = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) !=
null) {
        tempTwo = tempOne;
      };

      if (tempTwo != null) {
        variable = (EntryDouble) tempTwo;
        newData  = true;
      } // end of if statement
    }
    catch (Exception ex) {
      ex.printStackTrace();
    } // end of try-catch statement
  } // end of readFromSpace

} // end of NetworkDoubleSampledBuffer class
```

5. Float Type

```
/**
 * Title:        NetworkFloatSampledBuffer
 * Description:  This class implements a Sampled buffer using JavaSpace for Float values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import tuplespace.entries.*;
```

```
public class NetworkFloatSampledBuffer {

  private JavaSpace space;
  private EntryFloat variable;
  private Entry tempOne;
  private Entry tempTwo;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;



  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

    public EvtListener() throws RemoteException {
    } // end of EvtListener default constructor



    public void notify(RemoteEvent ev) {
      readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
public NetworkFloatSampledBuffer(String id, JavaSpace space, long latency) {
  this.space    = space;
  this.id       = id;
  this.latency  = latency;
  try {
     this.template = this.space.snapshot(new EntryFloat(id));
     this.space.write(new EntryFloat(), null, 6000);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  this.newData  = false;
} // end of constructor


/*
*  Constructor for state stream mode
*/
public NetworkFloatSampledBuffer(float x, String id, JavaSpace space, long latency) {
   this.variable = new EntryFloat(id, x);
  this.space    = space;
  this.id       = id;
  this.latency  = latency;

  try {
     this.template = this.space.snapshot(new EntryFloat(id));
     this.space.write(new EntryFloat(), null, 6000);
  }
```

```
catch (Exception e) {
      e.printStackTrace();
    }
    this.newData  = false;
  } // end of constructor
  ////////////////////////////////////////////////////////////////////////
  // public methods
  ////////////////////////////////////////////////////////////////////////
  /*
  * This method returns the current value
  */
  synchronized public float read() {
    newData = false;
    return variable.entryFloat.floatValue();
  } // end of read
 /*
  * This method writes the given value to the javaspace
  */
  synchronized public void write(float value) {
    EntryFloat entry = new EntryFloat(id, value);
    if (space != null) {
      try {
        space.write(entry, null, latency);
      }
      catch (RemoteException ex) {
        ex.printStackTrace();
      }
      catch (Exception ex) {
        ex.printStackTrace();
      } // end of try-catch statement
    } // end of if statement
  } // end of write
```

```
/*
* This method returns the newData value
*/
synchronized public boolean newData() {
    return newData;
} // end of newData



 /*
 * This method is used to enable notification
 */
 public void setNotification() {
  try {
     EvtListener listener = new EvtListener();
     if (space != null) {
        space.notify(template, null, listener, Long.MAX_VALUE, null);
     } // end of if statement
  }
  catch (Exception ex) {
     ex.printStackTrace();
  } // end of try-catch statement
 } // end of setNotification
```

```
  ////////////////////////////////////////////////////////////////////////
  // private methods
  ////////////////////////////////////////////////////////////////////////

  /*
  * This method is called by the notify method of EvtListener object
  * When called, it takes every object in the javaspace with predefined id
  */
  private void readFromSpace() {
    try {
      while((tempOne = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) !=
null) {
        tempTwo = tempOne;
      };

      if (tempTwo != null) {
        variable = (EntryFloat) tempTwo;
        newData  = true;
      } // end of if statement
    }
    catch (Exception ex) {
      ex.printStackTrace();
    } // end of try-catch statement
  } // end of readFromSpace
} // end of NetworkFloatSampledBuffer class
```

6. Long Type

```
/**
 * Title:        NetworkLongSampledBuffer
 * Description:  This class implements a Sampled buffer using JavaSpace for Long values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import tuplespace.entries.*;
```

```
public class NetworkLongSampledBuffe r {

  private JavaSpace space;
  private EntryLong variable;
  private Entry tempOne;
  private Entry tempTwo;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;



  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

    public EvtListener() throws RemoteException {
    } // end of EvtListener default constructor



    public void notify(RemoteEvent ev) {
      readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
public NetworkLongSampledBuffer(String id, JavaSpace space, long latency) {
  this.space    = space;
  this.id       = id;
  this.latency  = latency;
  try {
     this.template = this.space.snapshot(new EntryLong(id));
     this.space.write(new EntryLong(), null, 6000);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  this.newData  = false;
} // end of constructor


/*
*  Constructor for state stream mode
*/
public NetworkLongSampledBuffer(long x, String id, JavaSpace space, long latency) {
   this.variable = new EntryLong(id, x);
  this.space    = space;
  this.id       = id;
  this.latency  = latency;

  try {
     this.template = this.space.snapshot(new EntryLong(id));
     this.space.write(new EntryLong(), null, 6000);
  }
```

```
catch (Exception e) {
      e.printStackTrace();
    }
    this.newData  = false;
  } // end of constructor
  ////////////////////////////////////////////////////////////////////////
  // public methods
  ////////////////////////////////////////////////////////////////////////
  /*
  * This method returns the current value
  */
  synchronized public long read() {
    newData = false;
    return variable.entryLong.longValue();
  } // end of read
 /*
  * This method writes the given value to the javaspace
  */
  synchronized public void write(long value) {
    EntryLong entry = new EntryLong(id, value);
    if (space != null) {
      try {
         space.write(entry, null, latency);
      }
      catch (RemoteException ex) {
        ex.printStackTrace();
      }
      catch (Exception ex) {
        ex.printStackTrace();
      } // end of try-catch statement
    } // end of if statement
  } // end of write
```

```
/*
* This method returns the newData value
*/
synchronized public boolean newData() {
    return newData;
} // end of newData



 /*
 * This method is used to enable notification
 */
 public void setNotification() {
  try {
     EvtListener listener = new EvtListener();
     if (space != null) {
        space.notify(template, null, listener, Long.MAX_VALUE, null);
     } // end of if statement
  }
  catch (Exception ex) {
     ex.printStackTrace();
  } // end of try-catch statement
 } // end of setNotification
```

```
  ////////////////////////////////////////////////////////////////////////
  // private methods
  ////////////////////////////////////////////////////////////////////////

  /*
  * This method is called by the notify method of EvtListener object
  * When called, it takes every object in the javaspace with predefined id
  */
  private void readFromSpace() {
    try {
      while((tempOne = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) !=
null) {
        tempTwo = tempOne;
      };

      if (tempTwo != null) {
        variable = (EntryLong) tempTwo;
        newData  = true;
      } // end of if statement
    }
    catch (Exception ex) {
      ex.printStackTrace();
    } // end of try-catch statement
  } // end of readFromSpace
} // end of NetworkLongSampledBuffer class
```

7. HashMap Type

```
/**
 * Title:        NetworkHashSampledBuffer
 * Description:  This class implements a Sampled buffer using JavaSpace for Hashmap values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import tuplespace.entries.*;
import java.util.HashMap;
```

```
public class NetworkHashSampledBuffer {

  private JavaSpace space;
  private EntryHash variable;
  private Entry tempOne;
  private Entry tempTwo;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;


  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

    public EvtListener() throws RemoteException {
    } // end of EvtListener default constructor


    public void notify(RemoteEvent ev) {
      readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
 public NetworkHashSampledBuffer(String id, JavaSpace space, long latency) {
   this.space    = space;
   this.id       = id;
   this.latency  = latency;
   try {
      this.template = this.space.snapshot(new EntryHash(id));
      this.space.write(new EntryHash(), null, 6000);
   }
   catch (Exception e) {
     e.printStackTrace();
   }
   this.newData  = false;
 } // end of constructor



 /*
 *  Constructor for state stream mode
 */
  public NetworkHashSampledBuffer(HashMap x, String id, JavaSpace space, long
latency) {
    this.variable = new EntryHash(id, x);
   this.space    = space;
   this.id       = id;
   this.latency  = latency;


   try {
      this.template = this.space.snapshot(new EntryHash(id));
      this.space.write(new EntryHash(), null, 6000);
   }
```

```
    catch (Exception e) {
      e.printStackTrace();
    }
    this.newData  = false;
  } // end of constructor
  ////////////////////////////////////////////////////////////////////////
  // public methods
  ////////////////////////////////////////////////////////////////////////
  /*
  * This method returns the current value
  */
  synchronized public HashMap read() {
    newData = false;
    return variable.getHashMap();
  } // end of read
 /*
  * This method writes the given value to the javaspace
  */
  synchronized public void write(HashMap value) {
    EntryHash entry = new EntryHash(id, value);
    if (space != null) {
      try {
         space.write(entry, null, latency);
      }
      catch (RemoteException ex) {
        ex.printStackTrace();
      }
      catch (Exception ex) {
        ex.printStackTrace();
      } // end of try-catch statement
    } // end of if statement
  } // end of write
```

```
/*
* This method returns the newData value
*/
synchronized public boolean newData() {
    return newData;
} // end of newData



 /*
 * This method is used to enable notification
 */
 public void setNotification() {
  try {
     EvtListener listener = new EvtListener();
     if (space != null) {
        space.notify(template, null, listener, Long.MAX_VALUE, null);
     } // end of if statement
  }
  catch (Exception ex) {
     ex.printStackTrace();
  } // end of try-catch statement
 } // end of setNotification
```

```
  ////////////////////////////////////////////////////////////////////////
  // private methods
  ////////////////////////////////////////////////////////////////////////

  /*
  * This method is called by the notify method of EvtListener object
  * When called, it takes every object in the javaspace with predefined id
  */
  private void readFromSpace() {
    try {
      while((tempOne = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) !=
null) {
        tempTwo = tempOne;
      };

      if (tempTwo != null) {
        variable = (EntryHash) tempTwo;
        newData  = true;
      } // end of if statement
    }
    catch (Exception ex) {
      ex.printStackTrace();
    } // end of try-catch statement
  } // end of readFromSpace
} // end of NetworkHashSampledBuffer class
```

B. DATAFLOW NETWORK BUFFERS

1. String Type

```
/**
 * Title:        NetworkStringFIFOBuffer
 * Description:  This class implements a FIFO buffer using JavaSpace for string values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import java.util.Vector;
import tuplespace.entries.*;
```

```
public class NetworkStringFIFOBuffer {

  private JavaSpace space;
  private Vector variable;
  private Entry temp;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;


  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

     public EvtListener() throws RemoteException {
     } // end of EvtListener default constructor


    public void notify(RemoteEvent ev) {
       readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
public NetworkStringFIFOBuffer(String id, JavaSpace space, long latency) {
  this.space    = space;
  this.id       = id;
  this.variable = new Vector();
  this.latency  = latency;
  try {
     this.template = this.space.snapshot(new EntryString(id));
     this.space.write(new EntryString(), null, 6000);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  this.newData  = false;
} // end of constructor
////////////////////////////////////////////////////////////////////////////
// public methods
////////////////////////////////////////////////////////////////////////////

/*
* This method returns the first value in the queue
*/
synchronized public String read() {
  String d = ((EntryString) variable.elementAt(0)).entryString.toString();
  variable.remove(0);
  if (variable.size() == 0) {
     newData = false;
  } // end of if
  return d;
} // end of read
```

```
/*
* This method writes the given value to the javaspace
*/
synchronized public void write(String value) {
   EntryString entry = new EntryString(id, value);
  if (space != null) {
     try {
        space.write(entry, null, latency);
     }
     catch (RemoteException ex) {
       ex.printStackTrace();
     }
     catch (Exception ex) {
       ex.printStackTrace();
     } // end of try-catch statement
   } // end of if statement
 } // end of write




/*
* This method returns the newData value
*/
synchronized public boolean newData() {
    return newData;
} // end of newData
```

```
  /*
  * This method is used to enable notification
  */
  public void setNotification() {
   try {
      EvtListener listener = new EvtListener();
      if (space != null) {
         space.notify(template, null, listener, Long.MAX_VALUE, null);
      } // end of if statement
   }
   catch (Exception ex) {
     ex.printStackTrace();
   } // end of try-catch statement
  } // end of setNotification

////////////////////////////////////////// private methods ////////////////////////////////////////////////
  /*
  * This method is called by the notify method of EvtListener object
  * When called, it takes every object in the javaspace with predefined id
  */
  private void readFromSpace() {
    try {
       while((temp = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) != null) {
          variable.add((EntryString) temp);
          newData  = true;
       };
    }
    catch (Exception ex) {
      ex.printStackTrace();
    } // end of try-catch statement
  } // end of readFromSpace
} // end of NetworkStringFIFOBuffer class
```

2. Boolean Type

```
/**
 * Title:        NetworkBooleanFIFOBuffer
 * Description:  This class implements a FIFO buffer using JavaSpace for Boolean values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import java.util.Vector;
import tuplespace.entries.*;
```

```
public class NetworkBooleanFIFOBuffer {

  private JavaSpace space;
  private Vector variable;
  private Entry temp;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;


  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

     public EvtListener() throws RemoteException {
     } // end of EvtListener default constructor


    public void notify(RemoteEvent ev) {
       readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
public NetworkBooleanFIFOBuffer(String id, JavaSpace space, long latency) {
  this.space    = space;
  this.id       = id;
  this.variable = new Vector();
  this.latency  = latency;
  try {
     this.template = this.space.snapshot(new EntryBoolean(id));
     this.space.write(new EntryBoolean(), null, 6000);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  this.newData  = false;
} // end of constructor
////////////////////////////////////////////////////////////////////////////
// public methods
////////////////////////////////////////////////////////////////////////////

/*
* This method returns the first value in the queue
*/
synchronized public boolean read() {
  boolean d = ((EntryBoolean) variable.elementAt(0)).entryBoolean.booleanValue();
  variable.remove(0);
  if (variable.size() == 0) {
     newData = false;
  } // end of if
  return d;
} // end of read
```

```
/*
* This method writes the given value to the javaspace
*/
synchronized public void write(boolean value) {
   EntryBoolean entry = new EntryBoolean(id, value);
   if (space != null) {
      try {
         space.write(entry, null, latency);
      }
      catch (RemoteException ex) {
        ex.printStackTrace();
      }
      catch (Exception ex) {
        ex.printStackTrace();
      } // end of try-catch statement
    } // end of if statement
 } // end of write




/*
* This method returns the newData value
*/
synchronized public boolean newData() {
    return newData;
} // end of newData
```

```
  /*
  * This method is used to enable notification
  */
  public void setNotification() {
   try {
      EvtListener listener = new EvtListener();
      if (space != null) {
         space.notify(template, null, listener, Long.MAX_VALUE, null);
      } // end of  if statement
   }
   catch (Exception ex) {
     ex.printStackTrace();
   } // end of try-catch statement
  } // end of setNotification

  ////////////////////////////////////////////// private methods//////////////////////////////////////////////////
  /*
  * This method is called by the notify method of EvtListener object
  * When called, it takes every object in the javaspace with predefined id
  */
  private void readFromSpace() {
    try {
       while((temp = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) != null) {
          variable.add((EntryBoolean) temp);
          newData  = true;
       };
    }
    catch (Exception ex) {
       ex.printStackTrace();
    } // end of try-catch statement
  } // end of readFromSpace
} // end of NetworkBooleanFIFOBuffer class
```

3. Integer Type

```
/**
 * Title:        NetworkIntegerFIFOBuffer
 * Description:  This class implements a FIFO buffer using JavaSpace for Integer values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import java.util.Vector;
import tuplespace.entries.*;
```

```
public class NetworkIntegerFIFOBuffer {

  private JavaSpace space;
  private Vector variable;
  private Entry temp;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;


  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

    public EvtListener() throws RemoteException {
    } // end of EvtListener default constructor


    public void notify(RemoteEvent ev) {
      readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
public NetworkIntegerFIFOBuffer(String id, JavaSpace space, long latency) {
  this.space    = space;
  this.id       = id;
  this.variable = new Vector();
  this.latency  = latency;
  try {
     this.template = this.space.snapshot(new EntryInteger(id));
     this.space.write(new EntryInteger(), null, 6000);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  this.newData  = false;
} // end of constructor
////////////////////////////////////////////////////////////////////////////
// public methods
////////////////////////////////////////////////////////////////////////////
/*
* This method returns the first value in the queue
*/
synchronized public int read() {
  int d = ((EntryInteger) variable.elementAt(0)).entryInteger.intValue();
  variable.remove(0);

  if (variable.size() == 0) {
     newData = false;
  } // end of if
  return d;
} // end of read
```

```
/*
* This method writes the given value to the javaspace
*/
synchronized public void write(int value) {
   EntryInteger entry = new EntryInteger(id, value);
   if (space != null) {
     try {
        space.write(entry, null, latency);
     }
     catch (RemoteException ex) {
       ex.printStackTrace();
     }
     catch (Exception ex) {
       ex.printStackTrace();
     } // end of try-catch statement
   } // end of if statement
 } // end of write



/*
* This method returns the newData value
*/
synchronized public boolean newData() {
   return newData;
} // end of newData
```

```
  /*
  * This method is used to enable notification
  */
  public void setNotification() {
   try {
      EvtListener listener = new EvtListener();
      if (space != null) {
         space.notify(template, null, listener, Long.MAX_VALUE, null);
      } // end of if statement
   }
   catch (Exception ex) {
     ex.printStackTrace();
   } // end of try-catch statement
  } // end of setNotification

  ////////////////////////////////////////////private methods/////////////////////////////////////////////////////
  /*
  * This method is called by the notify method of EvtListener object
  * When called, it takes every object in the javaspace with predefined id
  */
  private void readFromSpace() {
    try {
       while((temp = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) != null) {
          variable.add((EntryInteger) temp);
          newData  = true;
       };
    }
    catch (Exception ex) {
       ex.printStackTrace();
    } // end of try-catch statement
  } // end of readFromSpace
} // end of NetworkIntegerFIFOBuffer class
```

4. Double Type

```
/**
 * Title:        NetworkDoubleFIFOBuffer
 * Description:  This class implements a FIFO buffer using JavaSpace for double values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */


import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import java.util.Vector;
import tuplespace.entries.*;
```

```
public class NetworkDoubleFIFOBuffer {

  private JavaSpace space;
  private Vector variable;
  private Entry temp;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;


  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

    public EvtListener() throws RemoteException {
    } // end of EvtListener default constructor


    public void notify(RemoteEvent ev) {
      readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
public NetworkDoubleFIFOBuffer(String id, JavaSpace space, long  latency) {
   this.space    = space;
   this.id       = id;
   this.variable = new Vector();
   this.latency  = latency;

   try {
      this.template = this.space.snapshot(new EntryDouble(id));
      this.space.write(new EntryDouble(), null, 6000);
   }
   catch (Exception e) {
     e.printStackTrace();
   }
   this.newData  = false;
} // end of constructor
////////////////////////////////////////////////////////////////////////////
// public methods
////////////////////////////////////////////////////////////////////////////
/*
* This method returns the first value in the queue
*/
synchronized public double read() {
   double d = ((EntryDouble) variable.elementAt(0)).entryDouble.doubleValue();
   variable.remove(0);
   if (variable.size() == 0) {
      newData = false;
   } // end of if
   return d;
} // end of read
```

```
/*
* This method writes the given value to the javaspace
*/
synchronized public void write(double value) {
  EntryDouble entry = new EntryDouble(id, value);
  if (space != null) {
    try {
      space.write(entry, null, latency);
    }
    catch (RemoteException ex) {
      ex.printStackTrace();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    } // end of try-catch statement
  } // end of if statement
 } // end of write




/*
* This method returns the newData value
*/
synchronized public boolean newData() {
   return newData;
} // end of newData
```

```
  /*
   * This method is used to enable notification
   */
   public void setNotification() {
    try {
       EvtListener listener = new EvtListener();
       if (space != null) {
          space.notify(template, null, listener, Long.MAX_VALUE, null);
       } // end of if statement
    }
    catch (Exception ex) {
      ex.printStackTrace();
    } // end of try-catch statement
   } // end of setNotification

////////////////////////////////////////////////// private methods//////////////////////////////////////////////////////
  /*
   * This method is called by the notify method of EvtListener object
   * When called, it takes every object in the javaspace with predefined id
   */
   private void readFromSpace() {
     try {
        while((temp = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) != null) {
          variable.add((EntryDouble) temp);
          newData  = true;
        };
     }
     catch (Exception ex) {
       ex.printStackTrace();
     } // end of try-catch statement
   } // end of readFromSpace
} // end of NetworkDoubleFIFOBuffer class
```

5. Long Type

```
/**
 * Title:        NetworkLongFIFOBuffer
 * Description:  This class implements a FIFO buffer using JavaSpace for Long values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import java.util.Vector;
import tuplespace.entries.*;
```

```
public class NetworkLongFIFOBuffer {

  private JavaSpace space;
  private Vector variable;
  private Entry temp;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;


  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

    public EvtListener() throws RemoteException {
    } // end of EvtListener default constructor


    public void notify(RemoteEvent ev) {
      readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
public NetworkLongFIFOBuffer(String id, JavaSpace space, long latency) {
  this.space    = space;
  this.id       = id;
  this.variable = new Vector();
  this.latency  = latency;
  try {
     this.template = this.space.snapshot(new EntryLong(id));
     this.space.write(new EntryLong(), null, 6000);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  this.newData  = false;
} // end of constructor
////////////////////////////////////////////////////////////////////////////
// public methods
////////////////////////////////////////////////////////////////////////////
/*
* This method returns the first value in the queue
*/
synchronized public long read() {
  long d = ((EntryLong) variable.elementAt(0)).entryLong.longValue();
  variable.remove(0);
  if (variable.size() == 0) {
     newData = false;
  } // end of if
  return d;
} // end of read
```

```
/*
* This method writes the given value to the javaspace
*/
synchronized public void write(long value) {
   EntryLong entry = new EntryLong(id, value);
   if (space != null) {
     try {
        space.write(entry, null, latency);
     }
     catch (RemoteException ex) {
       ex.printStackTrace();
     }
     catch (Exception ex) {
       ex.printStackTrace();
     } // end of try-catch statement
   } // end of if statement
 } // end of write




/*
* This method returns the newData value
*/
synchronized public boolean newData() {
   return newData;
} // end of newData
```

```
  /*
  * This method is used to enable notification
  */
  public void setNotification() {
    try {
       EvtListener listener = new EvtListener();
       if (space != null) {
          space.notify(template, null, listener, Long.MAX_VALUE, null);
       } // end of if statement
    }
    catch (Exception ex) {
      ex.printStackTrace();
    } // end of try-catch statement
  } // end of setNotification
  ////////////////////////////////////////// private methods//////////////////////////////////////////////////////
/*
  * This method is called by the notify method of EvtListener object
  * When called, it takes every object in the javaspace with predefined id
  */
  private void readFromSpace() {
     try {
        while((temp = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) != null) {
          variable.add((EntryLong) temp);
          newData  = true;
        };
     }
     catch (Exception ex) {
       ex.printStackTrace();
     } // end of try-catch statement
  } // end of readFromSpace
} // end of NetworkLongFIFOBuffer class
```

6. Float Type

```
/**
 * Title:        NetworkFloatFIFOBuffer
 * Description:  This class implements a FIFO buffer using JavaSpace for float values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import java.util.Vector;
import tuplespace.entries.*;
```

```
public class NetworkFloatFIFOBuffer {

  private JavaSpace space;
  private Vector variable;
  private Entry temp;
  private Entry template;
  private String id;
  private long latency;
  private boolean newData;


  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

    public EvtListener() throws RemoteException {
    } // end of EvtListener default construc tor


    public void notify(RemoteEvent ev) {
      readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
public NetworkFloatFIFOBuffer(String id, JavaSpace space, long latency) {
  this.space    = space;
  this.id       = id;
  this.variable = new Vector();
  this.latency  = latency;
  try {
     this.template = this.space.snapshot(new EntryFloat(id));
     this.space.write(new EntryFloat(), null, 6000);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  this.newData  = false;
} // end of constructor
////////////////////////////////////////////////////////////////////////////
// public methods
////////////////////////////////////////////////////////////////////////////

/*
* This method returns the first value in the queue
*/
synchronized public double read() {
  float d = ((EntryFloat) variable.elementAt(0)).entryFloat.floatValue();
  variable.remove(0);
  if (variable.size() == 0) {
     newData = false;
  } // end of if
  return d;
} // end of read
```

```
/*
* This method writes the given value to the javaspace
*/
synchronized public void write(float value) {
   EntryFloat entry = new EntryFloat(id, value);
   if (space != null) {
     try {
        space.write(entry, null, latency);
     }
     catch (RemoteException ex) {
       ex.printStackTrace();
     }
     catch (Exception ex) {
       ex.printStackTrace();
     } // end of try-catch statement
   } // end of if statement
 } // end of write




/*
* This method returns the newData value
*/
synchronized public boolean newData() {
   return newData;
} // end of newData
```

```
  /*
  * This method is used to enable notification
  */
  public void setNotification() {
    try {
       EvtListener listener = new EvtListener();
       if (space != null) {
          space.notify(template, null, listener, Long.MAX_VALUE, null);
       } // end of if statement
    }
    catch (Exception ex) {
      ex.printStackTrace();
    } // end of try-catch statement
  } // end of setNotification
  ////////////////////////////////////////////////// private methods//////////////////////////////////////////////
  /*
  * This method is called by the notify method of EvtListener object
  * When called, it takes every object in the javaspace with predefined id
  */
  private void readFromSpace() {
     try {
        while((temp = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) != null) {
          variable.add((EntryFloat) temp);
          newData  = true;
        };
     }
     catch (Exception ex) {
       ex.printStackTrace();
     } // end of try-catch statement
  } // end of readFromSpace
} // end of NetworkFloatFIFOBuffer class
```

7. HashMap Type

```
/**
 * Title:        NetworkHashFIFOBuffer
 * Description:  This class implements a FIFO buffer using JavaSpace for Hash values
 * Company:      NPGS
 * @author       Tolga DEMIRTAS
 * @version 1.0
 */

import net.jini.core.entry.Entry;
import net.jini.space.JavaSpace;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.entry.UnusableEntryException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;
import java.util.Vector;
import java.util.HashMap;
import tuplespace.entries.*;
```

```
public class NetworkHashFIFOBuffer {

  private JavaSpace space;
  private Vector variable;
  private Entry temp;
  private Entry template;
  private String id;
  private long latency;
  private int lastRead;
  private boolean newData;


  /*
  * Listener inner class for notifications
  */
  class EvtListener extends UnicastRemoteObject implements RemoteEventListener {

    public EvtListener() throws RemoteException {
    } // end of EvtListener default constructor


    public void notify(RemoteEvent ev) {
      readFromSpace();
    } // end of notify
  } // end of EvtListener inner class
```

```
/*
*  Constructor
*/
public NetworkHashFIFOBuffer(String id, JavaSpace space, long latency) {
  this.space    = space;
  this.id       = id;
  this.variable = new Vector();
  this.latency  = latency;
  try {
     this.template = this.space.snapshot(new EntryHash(id));
     this.space.write(new EntryHash(), null, 6000);
  }
  catch (Exception e) {
    e.printStackTrace();
  }

  this.newData  = false;
} // end of constructor
//////////////////////////////////////////////////////////////////////////
// public methods
//////////////////////////////////////////////////////////////////////////
/*
* This method returns the first value in the queue
*/
synchronized public HashMap read() {
  HashMap d = ((EntryHash) variable.elementAt(0)).getHashMap();
  variable.remove(0);
  if (variable.size() == 0) {
     newData = false;
  } // end of if
  return d;
} // end of read
```

```
/*
* This method writes the given value to the javaspace
*/
synchronized public void write(HashMap value) {
   EntryHash entry = new EntryHash(id, value);
   if (space != null) {
     try {
        space.write(entry, null, latency);
     }
     catch (RemoteException ex) {
       ex.printStackTrace();
     }
     catch (Exception ex) {
       ex.printStackTrace();
     } // end of try-catch statement
   } // end of if statement
 } // end of write




/*
* This method returns the newData value
*/
synchronized public boolean newData() {
    return newData;
} // end of newData
```

```
  /*
  * This method is used to enable notification
  */
  public void setNotification() {
   try {
      EvtListener listener = new EvtListener();
      if (space != null) {
         space.notify(template, null, listener, Long.MAX_VALUE, null);
      } // end of if statement
   }
   catch (Exception ex) {
     ex.printStackTrace();
   } // end of try-catch statement
  } // end of setNotification

////////////////////////////////////////////////// private methods//////////////////////////////////////////////////
  /*
  * This method is called by the notify method of EvtListener object
  * When called, it takes every object in the javaspace with predefined id
  */
  private void readFromSpace() {
    try {
       while((temp = space.takeIfExists(template, null, JavaSpace.NO_WAIT)) != null) {
          variable.add((EntryHash) temp);
          newData  = true;
       };
    }
    catch (Exception ex) {
       ex.printStackTrace();
    } // end of try-catch statement
  } // end of readFromSpace
} // end of NetworkHashFIFOBuffer class
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[BAN84] A. D. Bierrel, and B. J. Nelson, "Implementing Remote Procedure Calls," ACM Trans. Computer Systems 2, 1984, pp. 39 – 59.

[CAA98] J. C. A. de Almeida, "Software Architecture for Distributed Real-Time Embedded Systems," Master's Thesis, Naval Postgraduate School, Monterey, California, 1998.

[CAD97] David F. Carr, "CORBA and DCOM: How Each Works," http://internetworld.com/print/1997/03/4/software/cobra.html.

[CAK01] Lap-Sun Cheung, and Yu-Kwong Kwok, "A Quantitative Comparison of Load Balancing Approaches in Distributed Object Computing Systems," IEEE 25th Annual International Compsac, 2001, pp. 257 – 262.

[CDK96] G. Coulouris, J. Dollimore, and T. Kindberg, "Distributed Systems: Concepts and Design," Addison-Wesley, 1996.

[CLO00] Cloudscape, and Cloudscape Java Database, http://www.cloudscape.com.

[DCO97] Frank E. Redmond, "DCOM: Microsoft Distributed Component Object Model," IDG Books, 1997.

[DOU01] B. P. Douglass, "Doing Hard Time," Addison-Wesley, 2001.

[EDW01] W. Keith Edwards, "Core JINI," Second Edition, The Sun Microsystems Press, 2001.

[FHA99] E. Freeman, S. Hupfer and K. Arnold, "JavaSpaces: Principles, Patterns, and Practice," Addison-Wesley, 1999.

[GEL85] D. Gelernter, "Generative Communication in Linda," ACM Trans. Programming Languages and Systems, 7(1), Jan. 1985, pp. 80 – 112.

[KIN99] B. K. Kin, "A Simple Software Agents Framework for Building Distributed Applications," Master's Thesis, Naval Postgraduate School, Monterey, California, 1999.

[LAP93] A. P. Laplante, "Real-Time Systems Design and Analysis: An Engineer's Handbook," IEEE Press, 1993.

[LAS96] Luqi, and M. Shing, "Real-Time Scheduling for Software Prototyping," Journal of Systems Integration 6, 1996, pp. 41 – 72.

[LBM00] Luqi, V. Berzins, M. Shing, N. Nada, and C. Eagle, "Computer Aided Prototyping System (CAPS) for Heterogeneous Systems Development and Integration," Proceedings of the 2000 Command and Control Research and Technology Symposium, Naval Postgraduate School, Monterey, CA, June 26 – 28, 2000.

[LBS01] Luqi, V. Berzins, J. Ge, M. Shing, M. Auguston, B. Bryant, and B. Kin, "DCAPS – Architecture for Distributed Computer Aided Prototyping System," IEEE Computer Society Press, 2001, pp. 103 – 108.

[LBS00] Luqi, V. Berzins, M. Shing, R. Riehle, and J. Nogueira, "Evolutionary Computer Aided Prototyping System," IEEE Proceedings 34th International Conference on Technology of Object-Oriented Languages and Systems, 2000, pp. 363 – 372.

[LBY88] Luqi, V. Berzins, and R. T. Yeh, "A Prototyping Language for Real-Time Software," IEEE Transactions on Software Engineering 14(10), Oct. 1988, pp. 1409 – 1423.

[LUQ93] Luqi, "Real-Time Constraints in a Rapid Prototyping Language," Comput. Lang. 18(2), 1993, pp. 77 – 103.

[LUQ92] Luqi, "Computer Aided Prototyping for a Command-and-Control System using CAPS," IEEE Software, 9(1), Jan. 1992, pp. 56 – 67.

[NAK00] P. Niemeyer, and J. Knudsen, "Learning Java," O'Reilly, May 2000.

[ORB95] CORBA Overview, and The OMA Reference Model, OMG, http://www.infosys.tuwien.ac.at/Research/Corba/OMG/arch2.html, 1995.

[ORB91] The Object Management Group, "Common Object Request Broker: Architecture and Specification," OMG Document Number 91.12.1, 1991.

[RMI00] Sun Microsystems, Remote Method Invocation, Java 2 SDK Documentation, http://www.java.sun.com/products/j2se/1.3/docs/guide/rmi/.

[RWJ01] William J. Ray, "Object Model Driven Code Generation for the Enterprise," IEEE Computer Society Press, 2001, pp. 84 – 89.

[SAW01] M. Stang, and S. Whinston, "Enterprise Computing with Jini Technology," IT Professional 3(1), Jan-Feb 2001, pp. 33 – 38.

[SIN97] K. P. Sinha, "Distributed Operating Systems," IEEE Press, 1997.

[TAN95] A. S. Tanenbaum, "Distributed Operating Systems," Prentice-Hall, 1995

[TSP00] IBM, Tspaces, http://www.almaden.ibm.com/cs/Tspaces

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Deniz Kuvvetleri Komutanligi
   Personel Daire Baskanligi
   Bakanliklar
   Ankara, TURKEY

4. Deniz Harp Okulu Komutanligi
   Kutuphanesi
   Tuzla
   Istanbul, TURKEY

5. Chairman, Code CS
   Naval Postgraduate School
   Monterey, California 93943-5118

6. Dr Man-Tak Shing, CS/SH
   Naval Postgraduate School
   Monterey, California 93943-5118

7. LTC Joseph Puett
   Naval Postgraduate School
   Monterey, California 93943-5118

8. LTJG Tolga Demirtas
   Deniz Harp Okulu Komutanligi
   Tuzla
   Istanbul, TURKEY